Chương 3: Tầng giao vận

Mục tiêu :

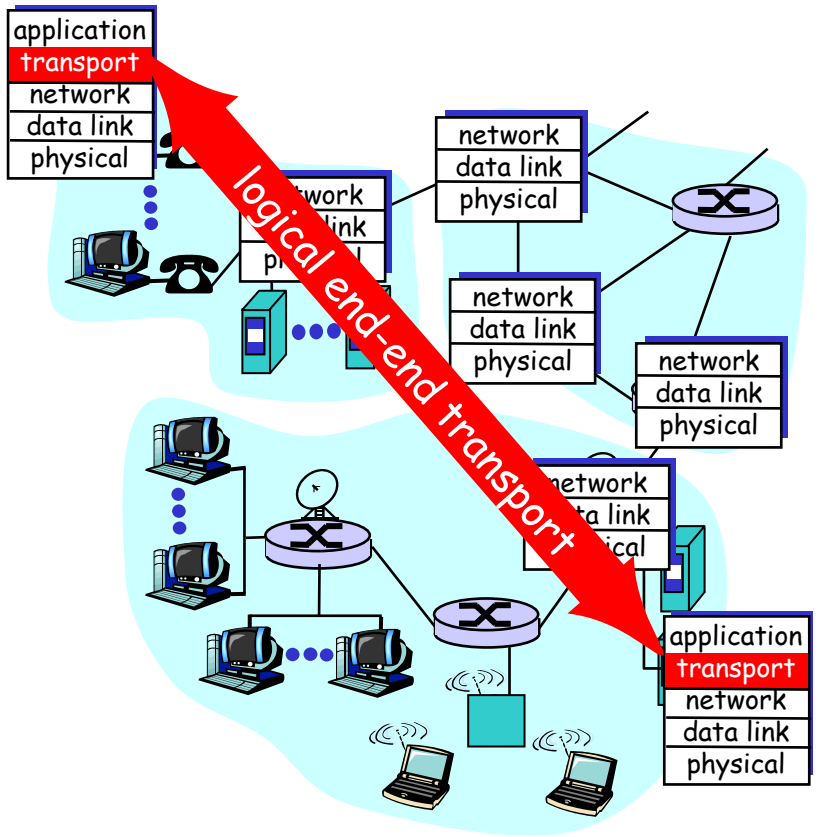
- Nguyên lý các dịch vụ của tầng giao vận:
  - Phân kênh / Dồn kênh
  - Xây dựng đường truyền tin cậy
  - Điều khiển lưu lượng
  - Kiểm soát tắc nghẽn
- Cài đặt những nguyên lý này trên Internet như thế nào ?

Sẽ học cái gì ?

- Các dịch vụ của tầng giao vận
- Phân kênh / Dồn kênh
- UDP: Giao thức không hướng nối
- Nguyên lý xây dựng đường truyền tin cậy
- TCP: Giao thức hướng nối
  - Tin cậy
  - Điều khiển lưu lượng
  - Kiểm soát tắc nghẽn
- Nguyên lý kiểm soát tắc nghẽn
- Kiểm soát tắc nghẽn trong TCP

Dịch vụ và Giao thức ở tầng Giao vận

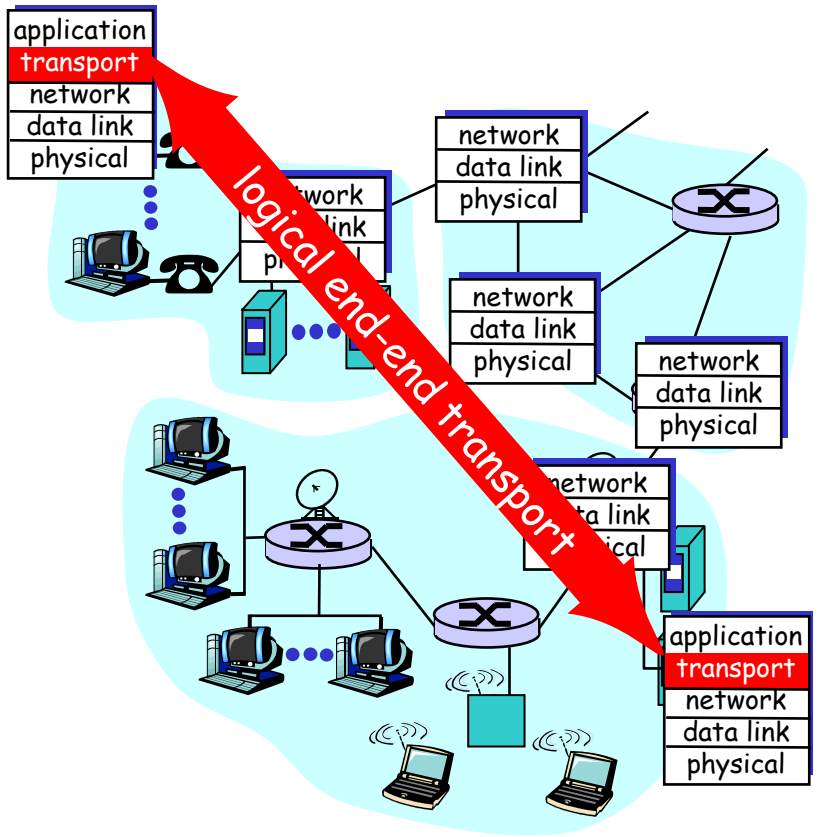
- Cung cấp *đường truyền luận lý* giữa các tiến trình chạy trên các thiết bị đầu cuối.
- Giao thức Giao vận chạy trên thiết bị đầu cuối
- Phân biệt dịch vụ của tầng Giao vận với tầng Mạng:
- *Tầng Mạng:* Chuyển dữ liệu giữa các thiết bị đầu cuối.
- *Tầng Giao vận:* Chuyển dữ liệu giữa các tiến trình
  - Dựa trên tầng Mạng nhưng biến đổi, tăng cường



Giao thức ở tầng Giao vận

Có hai giao thức giao vận trên Internet:

- Tin cậy, nhận theo đúng thứ tự gửi, một người nhận (TCP)
  - Kiểm soát tắc nghẽn
  - Điều khiển lưu lượng
  - Thiết lập đường truyền
- Không tin cậy ("cố gắng tối đa"), Không theo đúng thứ tự: UDP
- Cả TCP và UDP đều không cung cấp:
  - Chuyển theo thời gian thực
  - Băng thông tối thiểu
  - Nhiều người nhận tin cậy

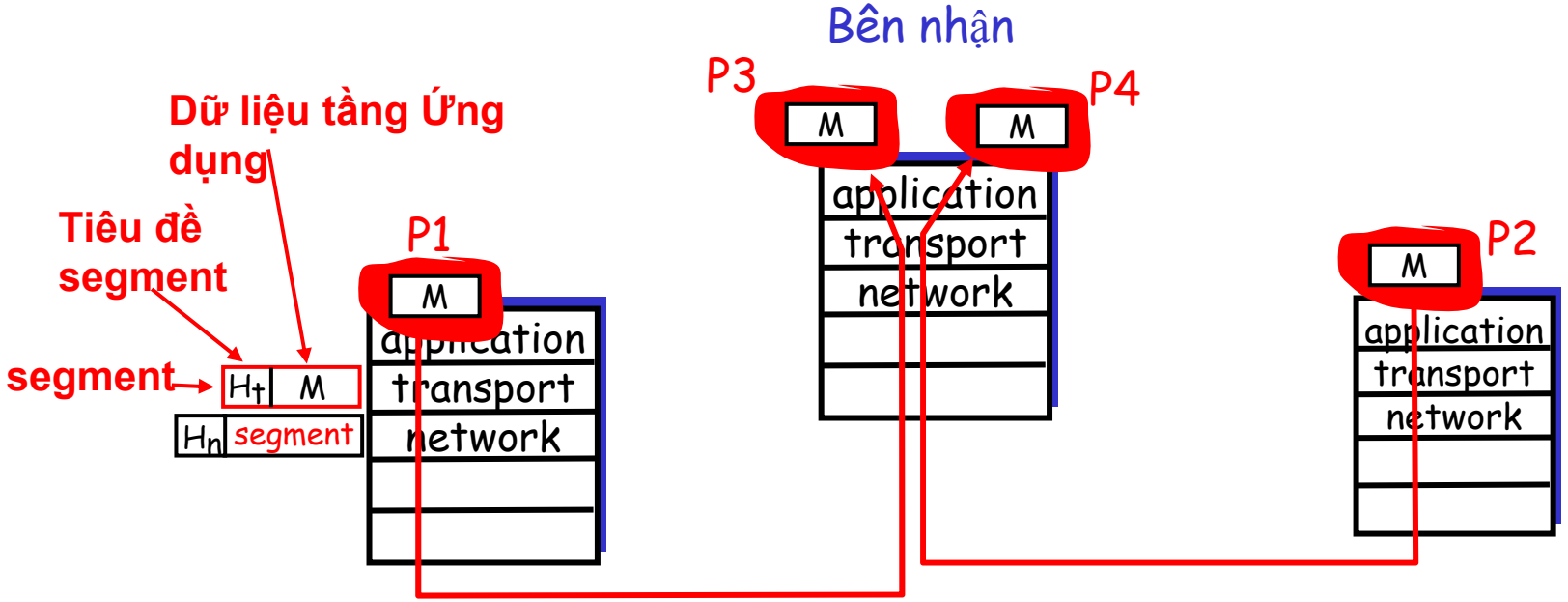


Phân kênh / Dồn kênh

Nhớ lại: *segment* – Đơn vị dữ liệu trao đổi giữa hai thực thể Giao vận

- Còn gọi là TPDU: Transport Protocol Data Unit

Phân kênh: Chuyển các segment nhận được cho Ứng dụng phù hợp

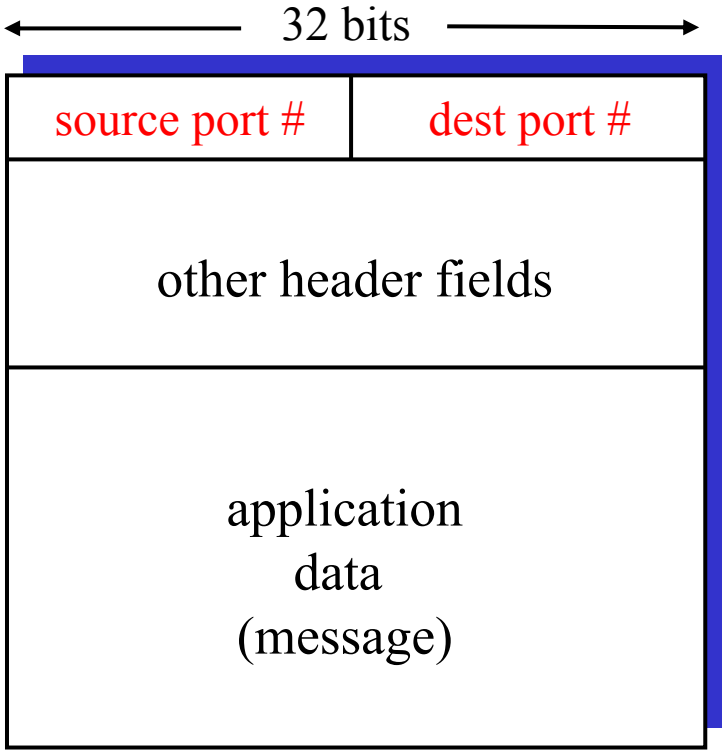


Phân kênh / Dồn kênh

Dồn kênh:
Lấy dữ liệu từ nhiều trình Ứng dụng, bổ sung thêm các tiêu đề (để sau này Phân kênh sử dụng)

Dồn kênh/Phân kênh:

- Dựa trên Địa chỉ IP, Số hiệu cổng của bên Gửi và bên Nhận
  - source, dest. port trong mỗi segment
  - Nhớ lại: Các số hiệu cổng của các ứng dụng thông dụng



Khuôn dạng tổng quát của TCP/UDP

Phân biệt Phân kênh và Dồn kênh

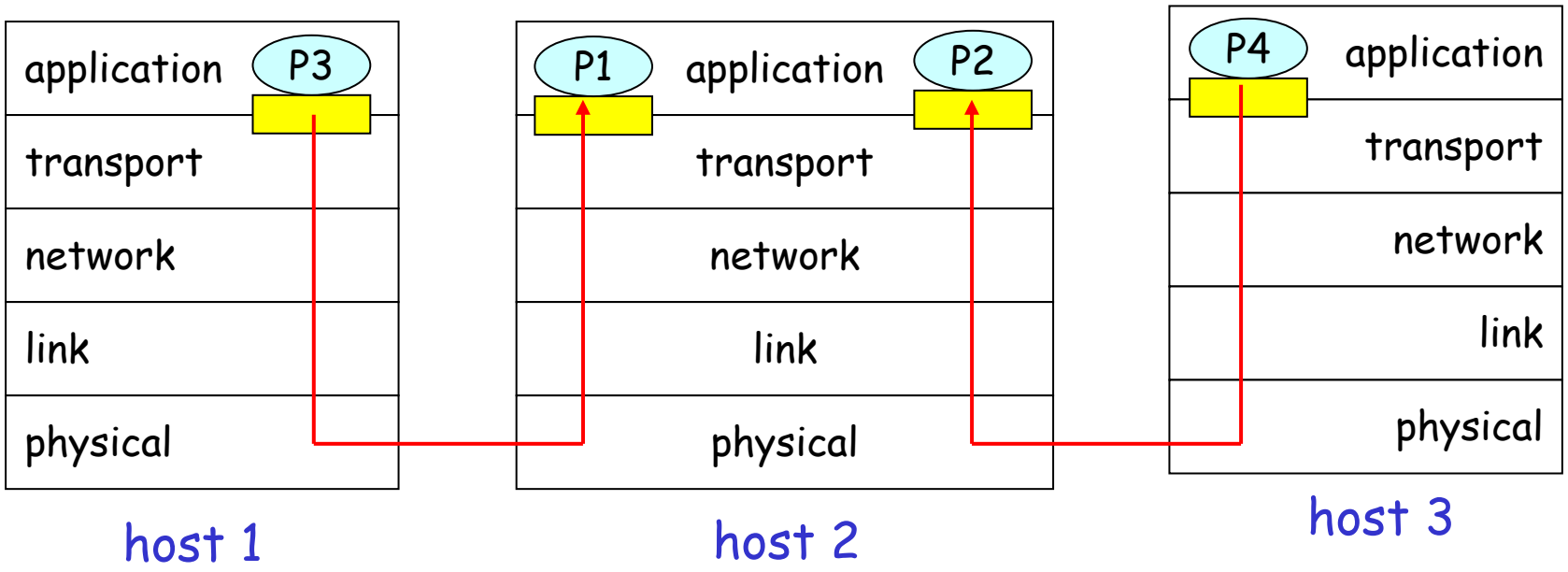
Phân kênh ở nút nhận:

Chuyển segment đến đúng socket

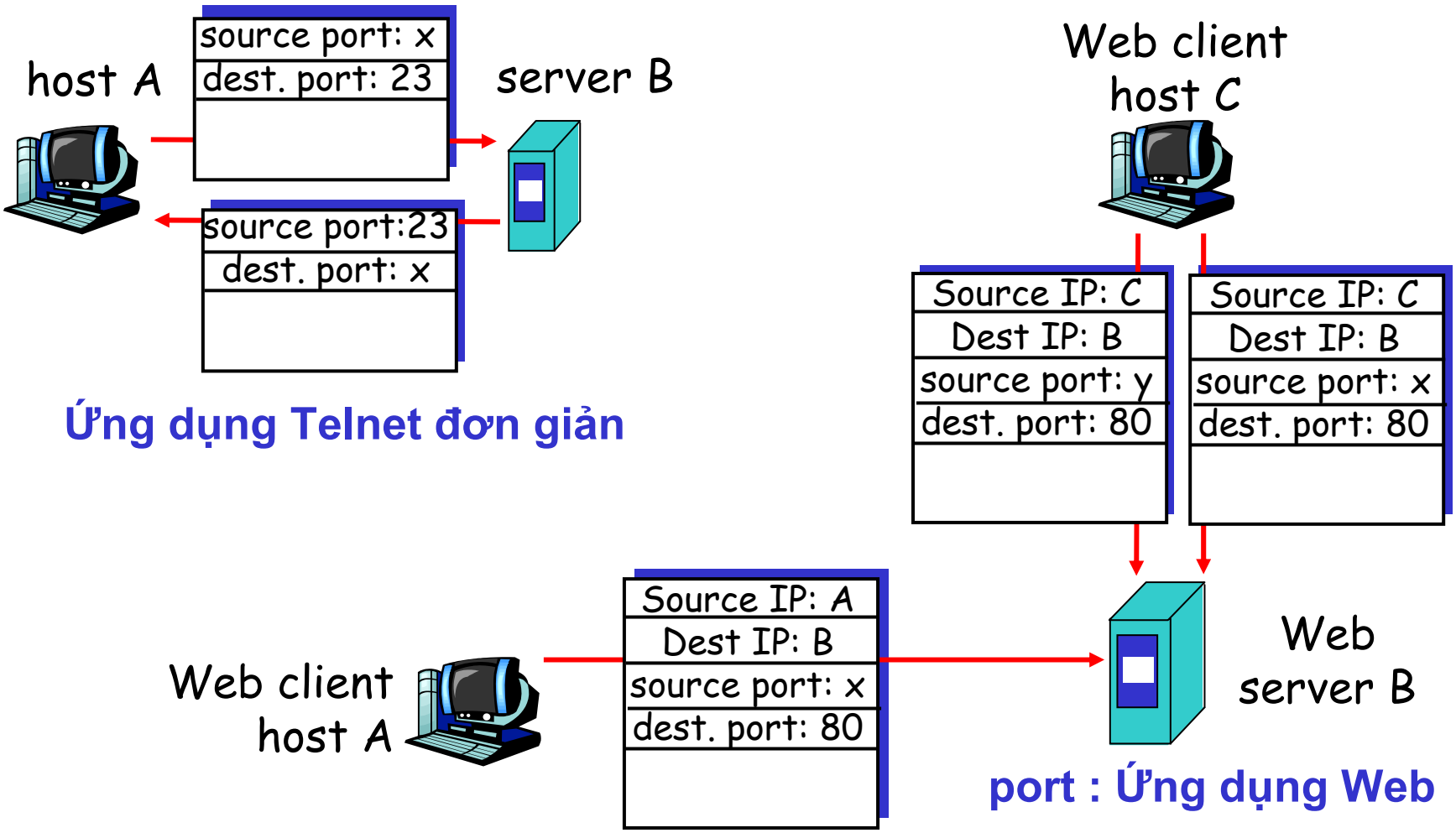
Dồn kênh ở nút gửi

Lấy dữ liệu từ nhiều socket, đặt trong các phong bì khác nhau (để phân kênh sau này)

= socket = process

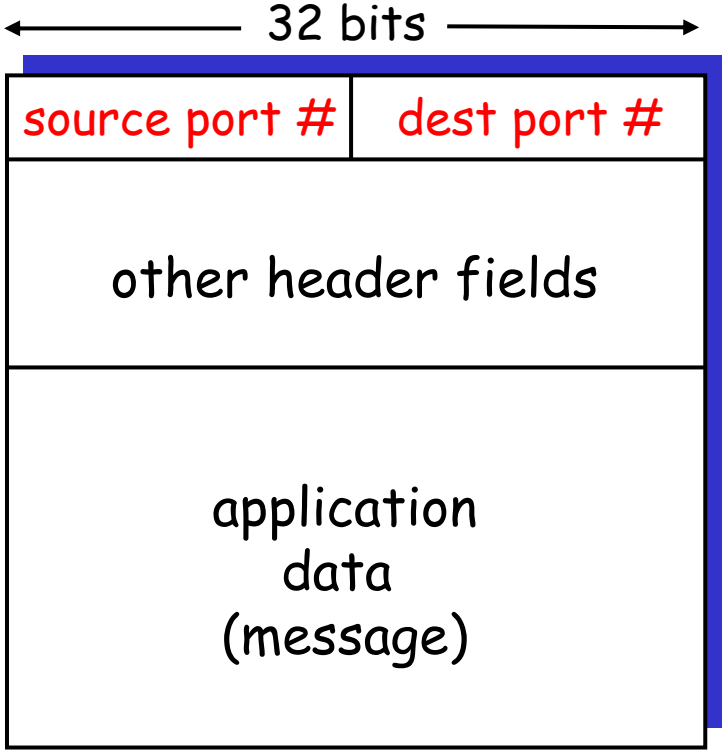


Ví dụ : Phân kênh / Dồn kênh



Phân kênh bằng cách nào ?

- Máy tính đích nhận được IP datagrams
  - Mỗi datagram có địa chỉ IP đích, IP nguồn
  - Mỗi datagram chứa một segment của tầng giao vận
  - Mỗi segment chứa địa chỉ cổng gửi, cổng nhận
    (chú ý : cổng của các ứng dụng thông dụng)
- Máy tính đích sử dụng địa chỉ IP và cổng của bên gửi để chuyển segment đến socket thích hợp



TCP/UDP segment format

UDP: User Datagram Protocol [RFC 768]

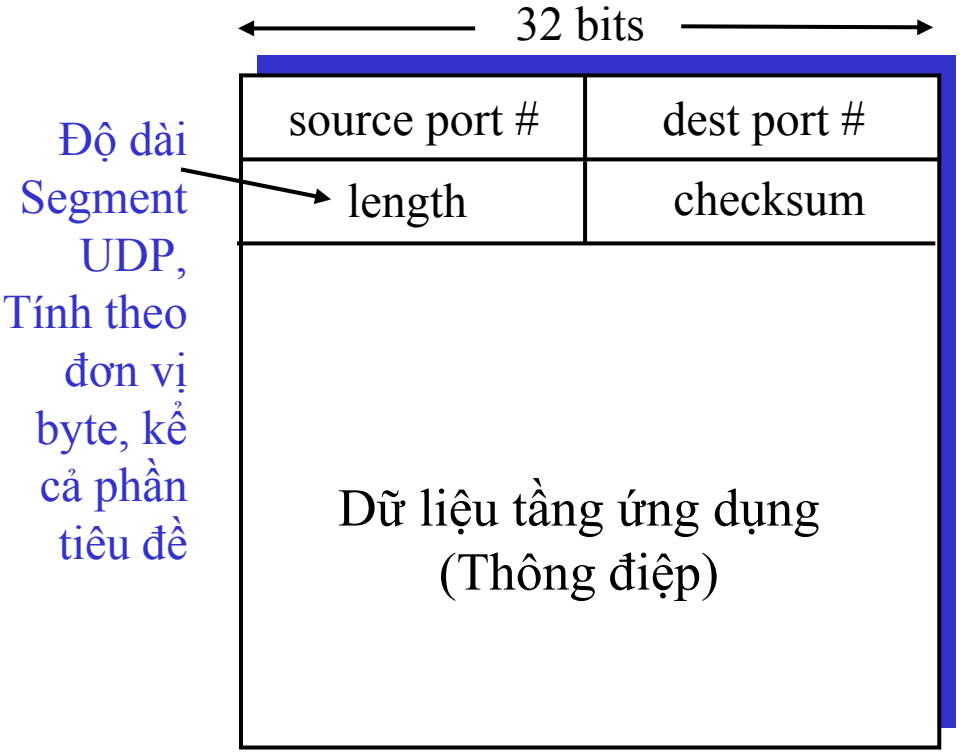
- Giao thức giao vận cực kỳ đơn giản, gần như không có gì
- Dịch vụ kiểu "cố gắng tối đa", các segment của UDP có thể :
  - Mất.
  - Tầng ứng dụng chịu trách nhiệm sắp xếp theo đúng thứ tự.
- ***Không hướng nối***:
  - Phía gửi, phía nhận không cần "bắt tay"
  - Các segment UDP được xử lý độc lập

Tại sao sử dụng UDP?

- Độ trễ nhỏ: Không có giai đoạn thiết lập đường truyền
- Đơn giản: Phía gửi và nhận không phải ghi nhớ trạng thái gửi/ nhận.
- Tiêu đề gói tin bé
- Không có cơ chế kiểm soát tắc nghẽn: Bên gửi có thể gửi dữ liệu với tốc độ tối đa.

UDP (tiếp)

- Thường được các ứng dụng đa phương tiện sử dụng
  - Chấp nhận mất dữ liệu
  - Tốc độ truyền quan trọng
- Một số ứng dụng khác cũng sử dụng (Vì sao?):
  - DNS
  - SNMP
- Muốn truyền tin cậy bằng UDP: phải đặt cơ chế tin cậy tại ứng dụng
  - Ứng dụng chịu trách nhiệm phát hiện và khắc phục lỗi!



Khuôn dạng segment UDP

UDP checksum

Mục tiêu: phát hiện "lỗi" (bit bị thay đổi giá trị) trong segment nhận được.

Bên gửi:

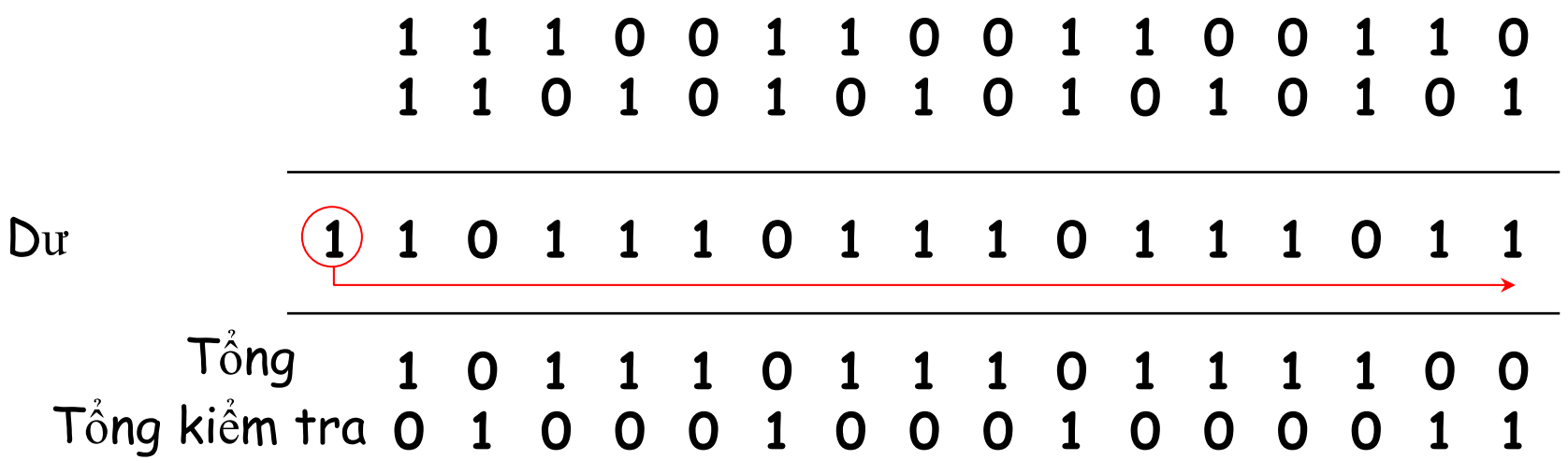
- Xem nội dung segment là các số nguyên 16-bit liên tiếp (các từ).
- checksum: tổng bù 1 của các từ
- Phía gửi sẽ đặt giá trị checksum tính được vào trường checksum trong tiêu đề gói tin UDP

Bên nhận:

- Tính giá trị checksum của segment vừa nhận được
- Kiểm tra xem giá trị vừa tính được có trùng với giá trị trong trường checkum ở tiên đề không:
  - **KHÔNG TRÙNG** – có lỗi
  - **TRÙNG** – Không phát hiện được lỗi. *Nhưng có thể có lỗi.*

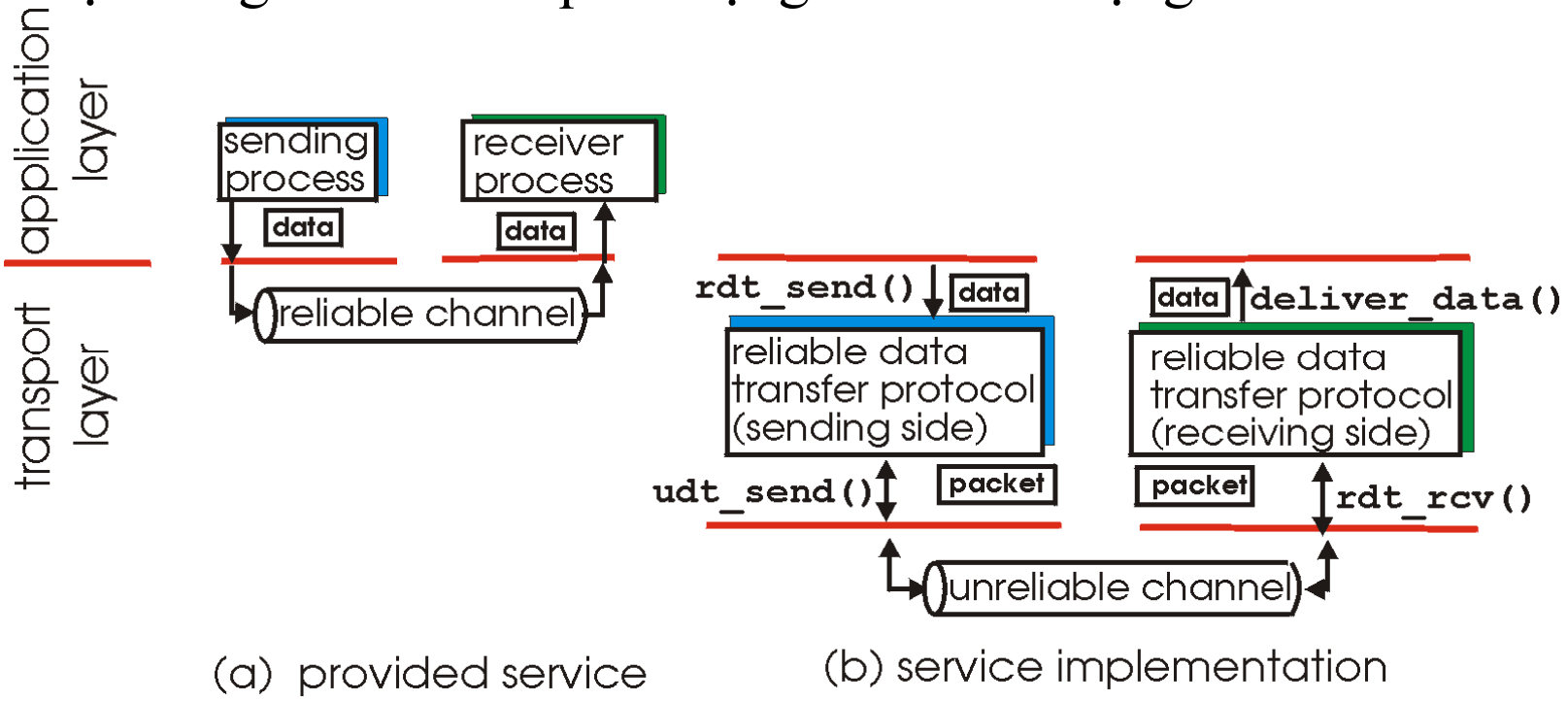
Ví dụ tính UDP Checksum

- Lưu ý:
  - Khi cộng hai số, nhớ ở bit cao nhất sẽ được cộng vào kết quả
- Ví dụ: Cộng hai số nguyên 16-bit



Nguyên lý Truyền tin cậy

- Có thể áp dụng trong Tầng Giao vận và Liên kết dữ liệu
- Một trong 10 vấn đề quan trọng nhất của Mạng!

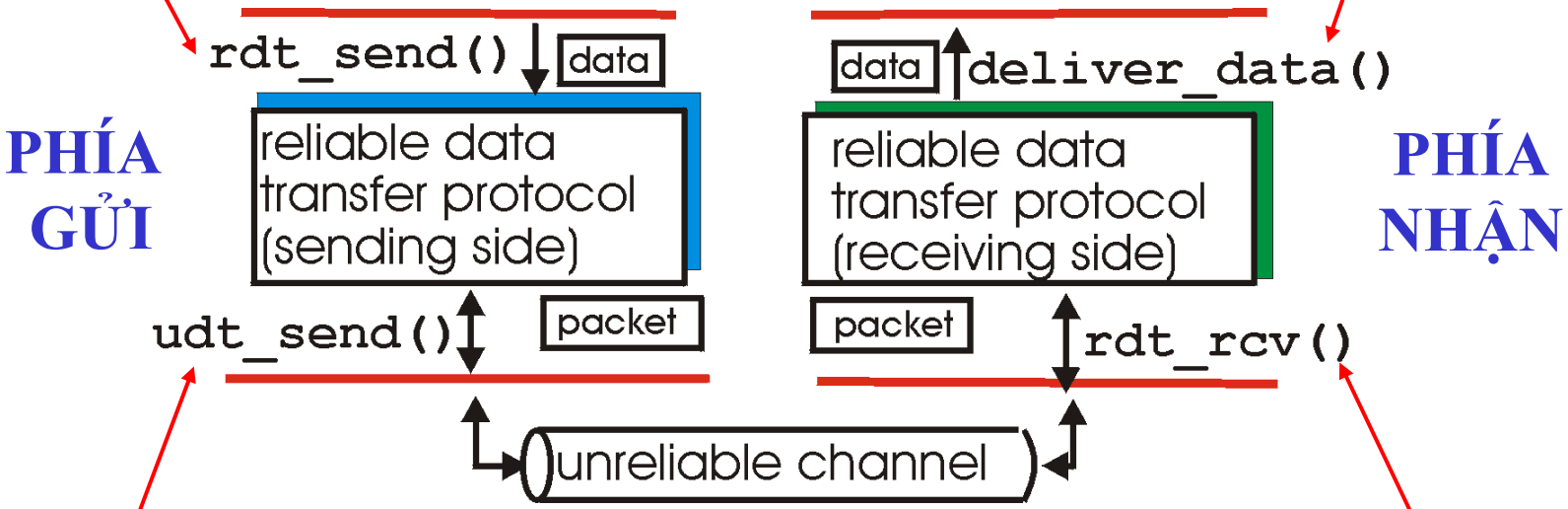


- Các đặc điểm của đường truyền không tin cậy phía dưới sẽ quyết định độ phức tạp của giao thức Truyền tin cậy (rdt)

Truyền tin cậy: Bắt đầu như thế nào ?

rdt_send(): Tầng trên gọi, (ví dụ Ứng dụng.). Yêu cầu chuyển dữ liệu tin cậy cho đối tác bên kia

deliver_data(): Rdt gọi để chuyển dữ liệu lên trên



udt_send(): rdt gọi để chuyển packet cho phía nhận qua đường truyền không tin cậy

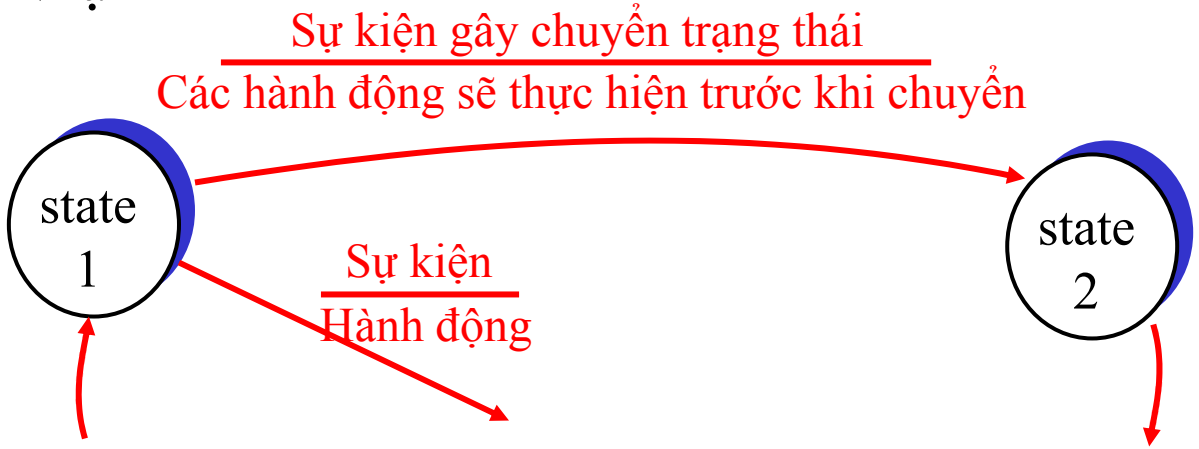
rdt_rcv(): được gọi khi packet đến phía nhận

Truyền tin cậy: Khởi đầu

Chúng ta sẽ :

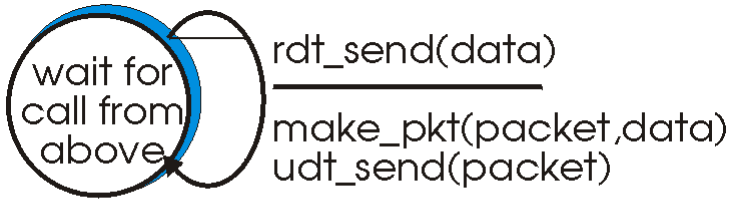
- Phát triển dần dần cả phía Gửi và phía Nhận trong Giao thức Truyền tin cậy (rdt)
- Xét dữ liệu chỉ truyền trên một hướng
  - Nhưng thông tin điều khiển có thể truyền theo cả hai hướng!
- Sử dụng máy Hữu hạn trạng thái (FSM) để miêu tả phía Gửi và phía Nhận

Trạng thái: Chỉ thay đổi sang trạng thái khác khi có sự kiện mang tính duy nhất xảy ra

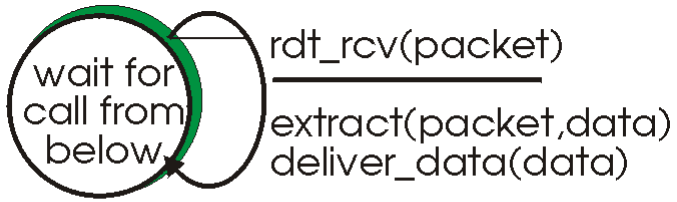


Rdt1.0: Truyền tin cậy qua kênh truyền tin cậy

- Đặc điểm của kênh truyền tin cậy
  - Bit trong gói tin không bị lỗi
  - Gói tin không bị mất
- FSM cho bên Gửi và bên Nhận độc lập nhau:
  - Phía Gửi truyền gói tin qua kênh truyền phía dưới
  - Phía Nhận đọc gói tin từ kênh truyền bên dưới



(a) rdt1.0: sending side

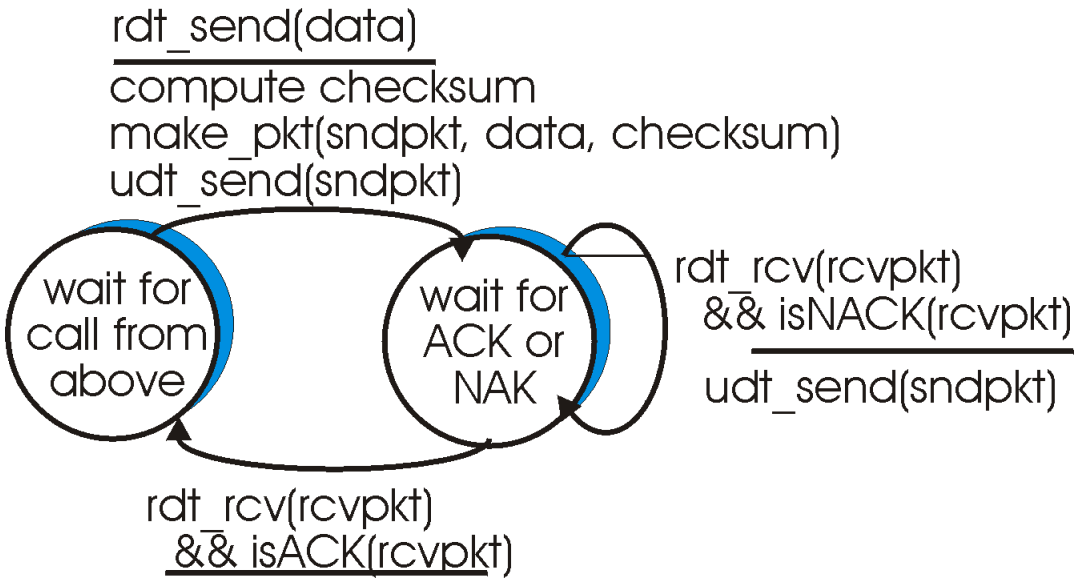


(b) rdt1.0: receiving side

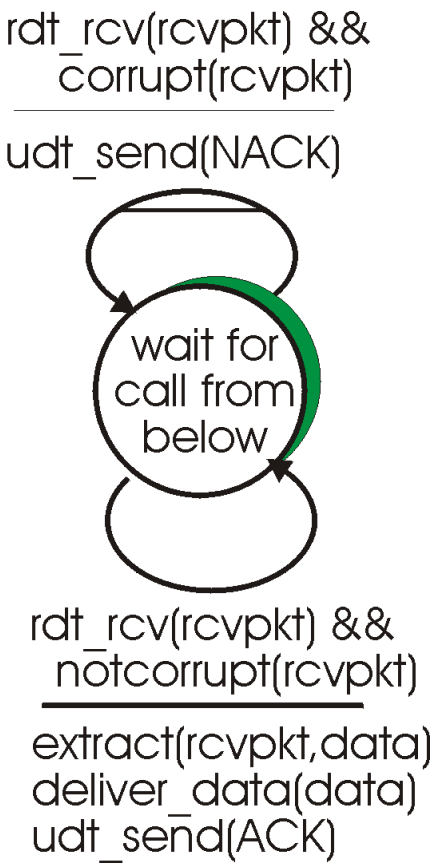
Rdt2.0: Kênh truyền có lỗi bit

- Kênh truyền bên dưới có thể khiến **bit trong gói tin lỗi**
  - Ghi nhớ: checksum có thể được sử dụng để phát hiện lỗi
- ***Vấn đề***: Khắc phục lỗi như thế nào ?
  - ***Biên nhận tích cực (ACK):*** Bên nhận thông báo tường minh cho bên gửi mình nhận đúng và chính xác gói tin.
  - ***Biên nhận tiêu cực (NACK):*** Bên nhận thông báo tường minh cho bên gửi gói tin mình nhận có lỗi.
  - Phía gửi gửi lại gói tin khi nhận được NACK
  - Ví dụ nào trong cuộc sống sử dụng ACK, NACK?
- Bổ sung thêm cơ chế mới trong **`rdt2.0`** :
  - Phát hiện lỗi
  - Phản hồi tường minh: thông điệp phản hồi (ACK,NACK) ***Nhận → Gửi***

rdt2.0: Đặc tả FSM

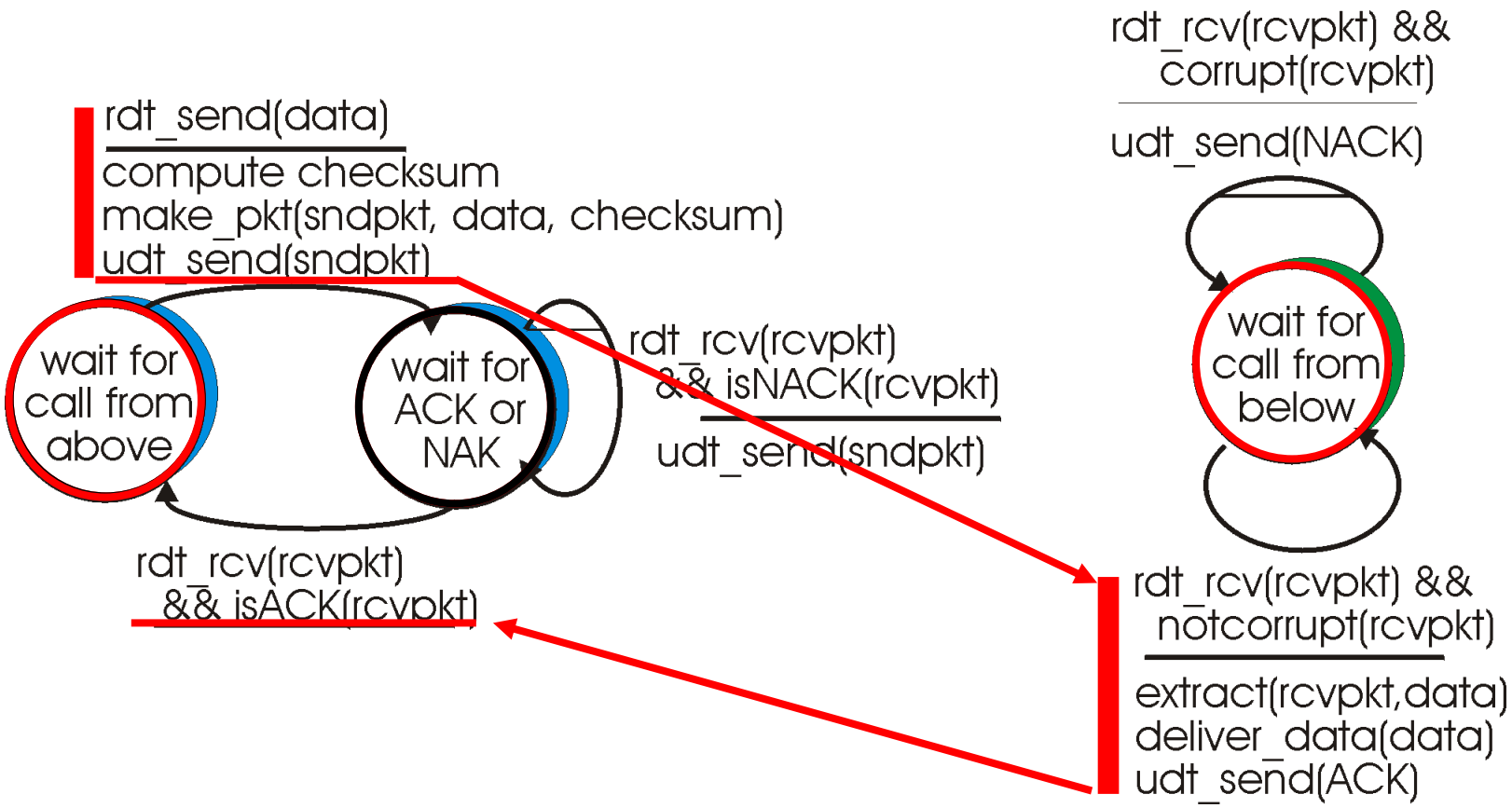


FSM Gửi



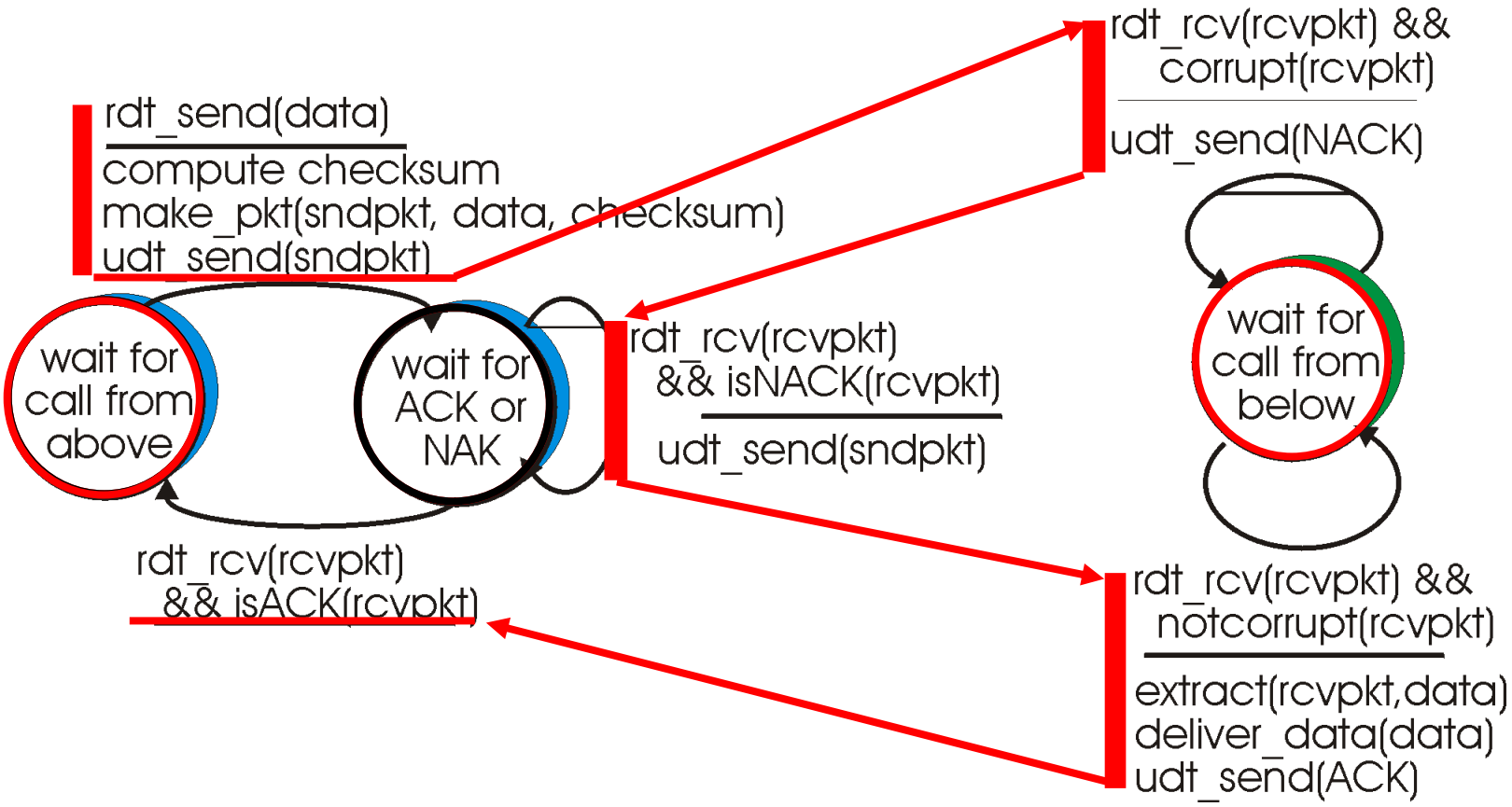
FSM Nhận

rdt2.0: Ví dụ không lỗi



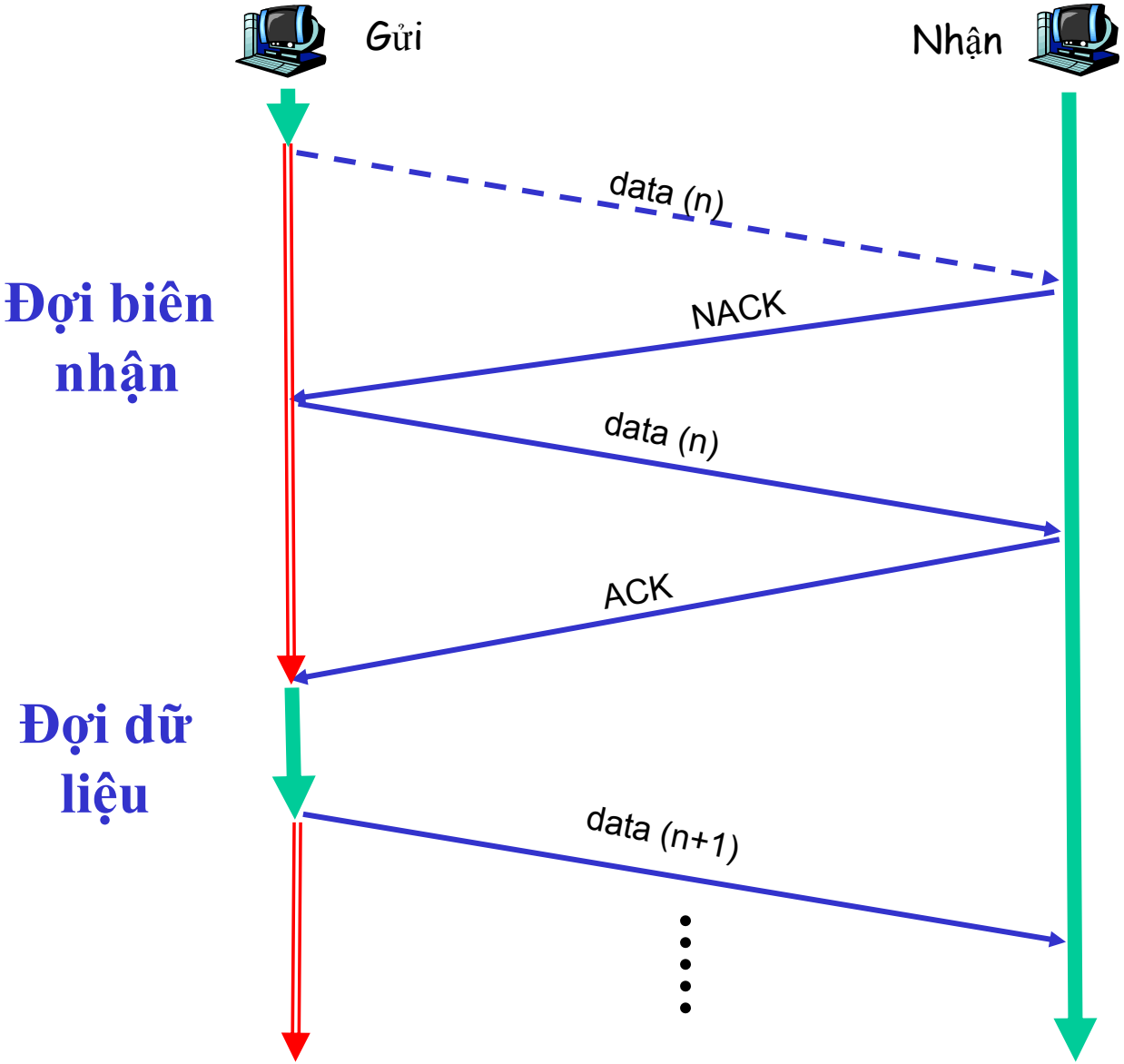
FSM Gửi **FSM Nhận**

rdt2.0: Ví dụ có lỗi



FSM Gửi **FSM Nhận**

Tổng quan về rdt2.0



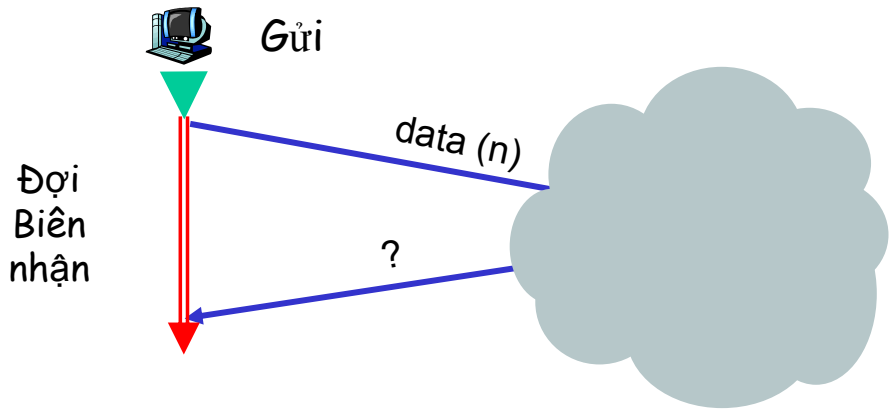
Đợi biên nhận

Đợi dữ liệu

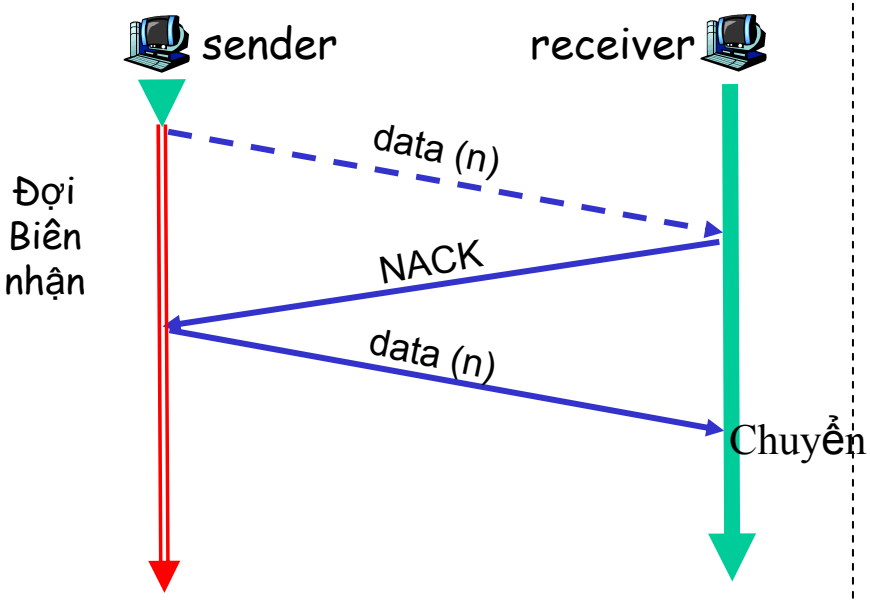
rdt2.0 chưa hoàn chỉnh !

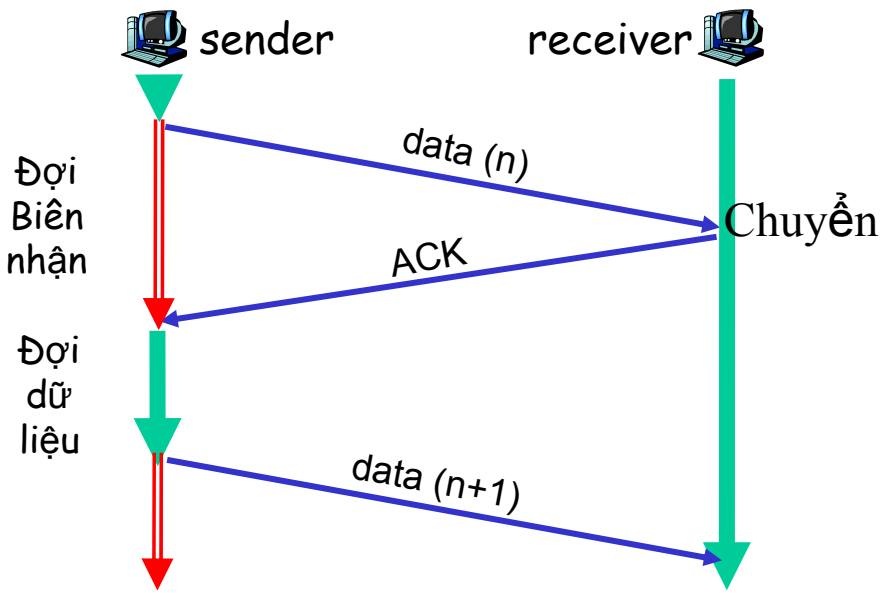
Chuyện gì xảy ra nếu ACK/NACK bị lỗi?

- Bên Gửi không nhận được phản hồi, nên không biết chuyện gì xảy ra ở phía Nhận !



Có hai trường hợp





rdt2.0 Có lỗi nghiêm trọng !

Như vậy nếu gói tin ACK/NACK bị lỗi

- Phía Gửi không biết gì về trạng thái phía Nhận !
- Nếu truyền lại: Dữ liệu có thể bị *trùng lặp*

Phải làm sao ?

- ***Phía gửi lại biên nhận cho phản hồi từ phía nhận?*** Nếu chính gói phản hồi bị mất thì sao ?
- ***Truyền lại ?*** Có thể truyền lại gói tin đã được nhận đúng !

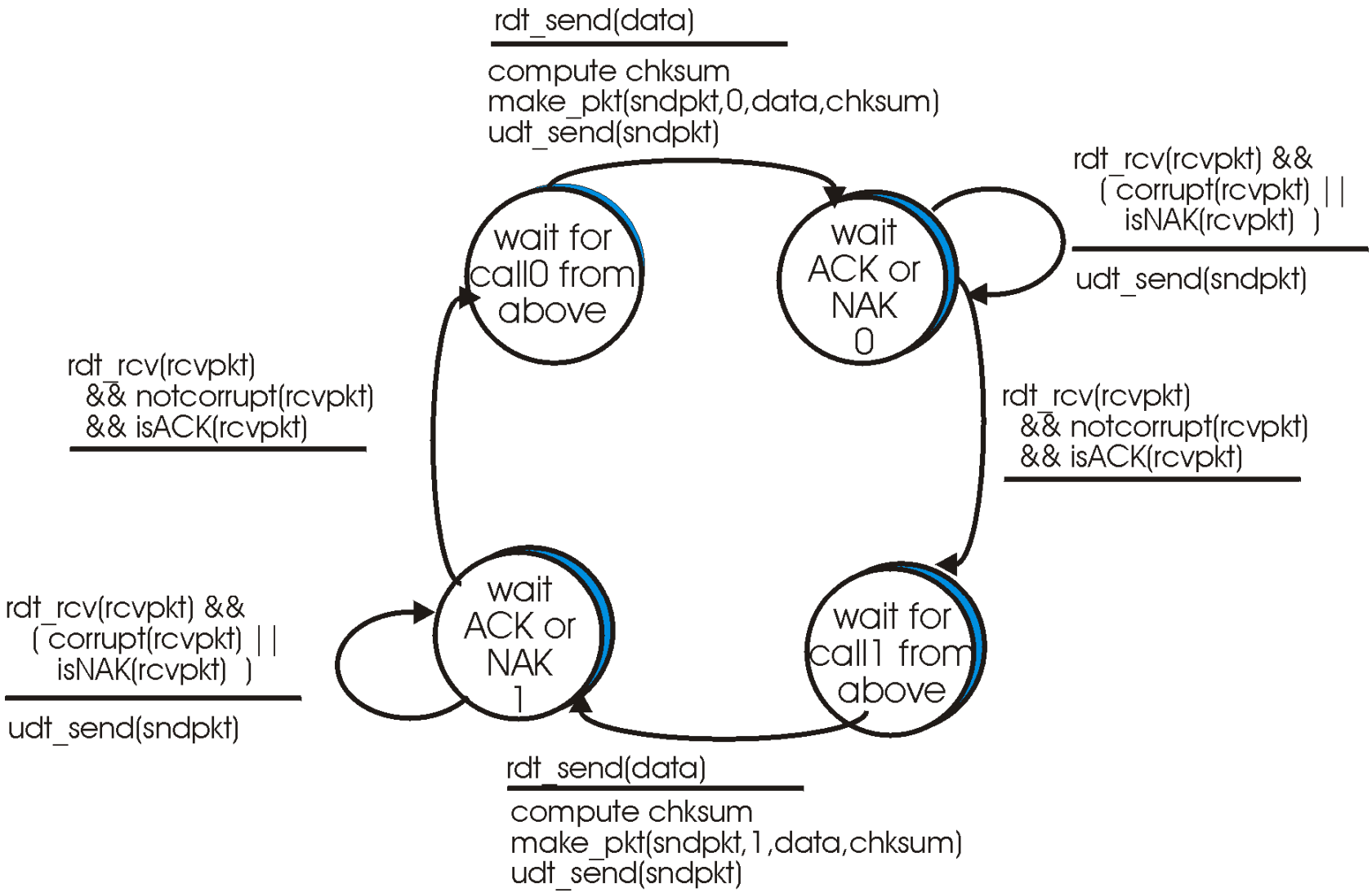
Xử lý trùng lặp ?

- Phía gửi đánh ***Số thứ tự*** cho mỗi gói tin gửi đi.
- Nếu gói tin phản hồi bị lỗi: phía Gửi truyền lại gói tin.
- Phía Nhận loại bỏ các gói tin trùng lặp (không chuyển lên trên)

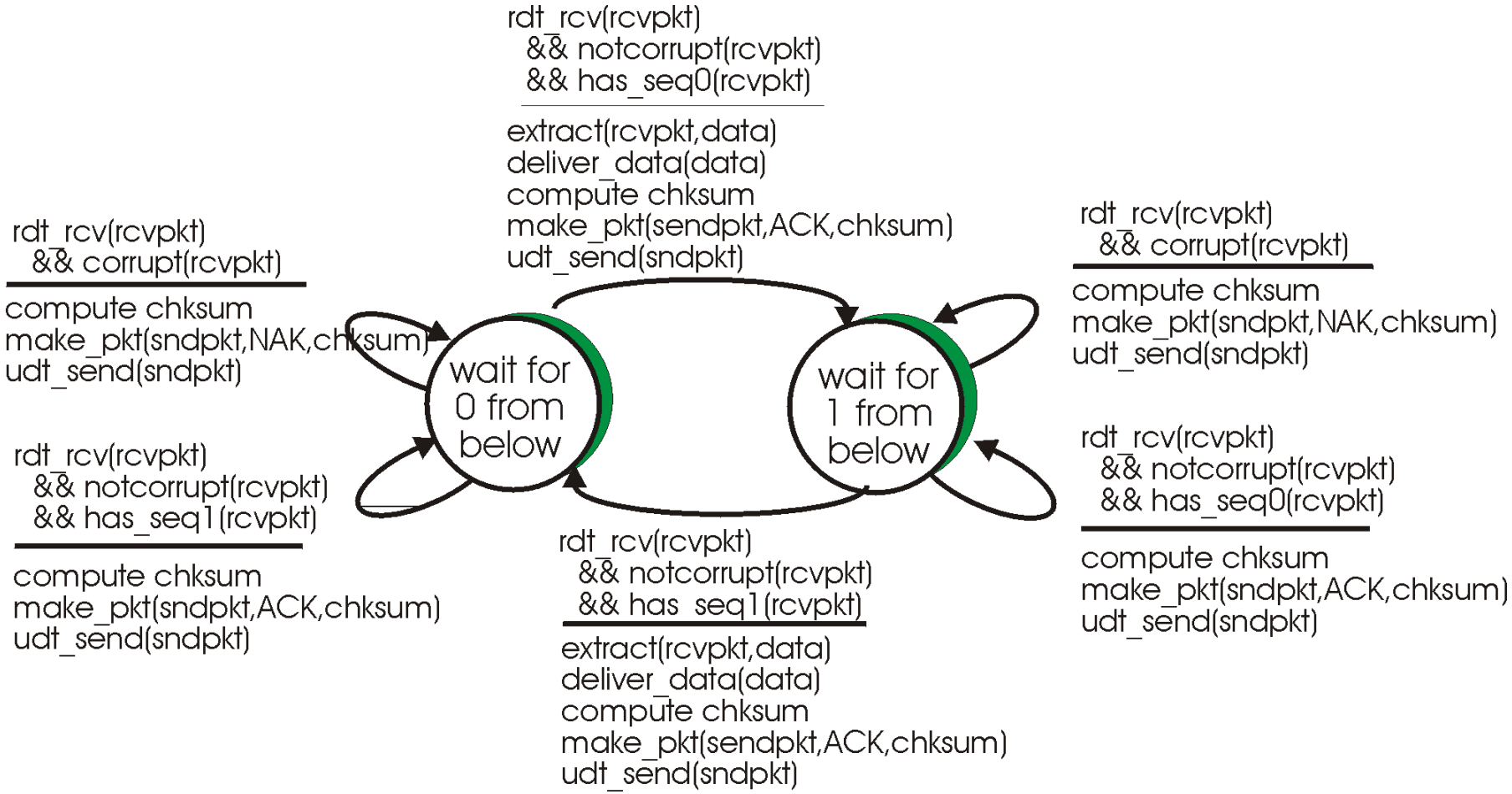
Dừng và Chờ

*Phía Gửi gửi một gói tin, **Dừng** lại và và **Chờ** phía nhận phản hồi.*

rdt2.1: Khắc phục Phản hồi bị lỗi (bên Gửi)



rdt2.1: Khắc phục Phản hồi bị lỗi (bên Nhận)



rdt2.1: Thảo luận ?

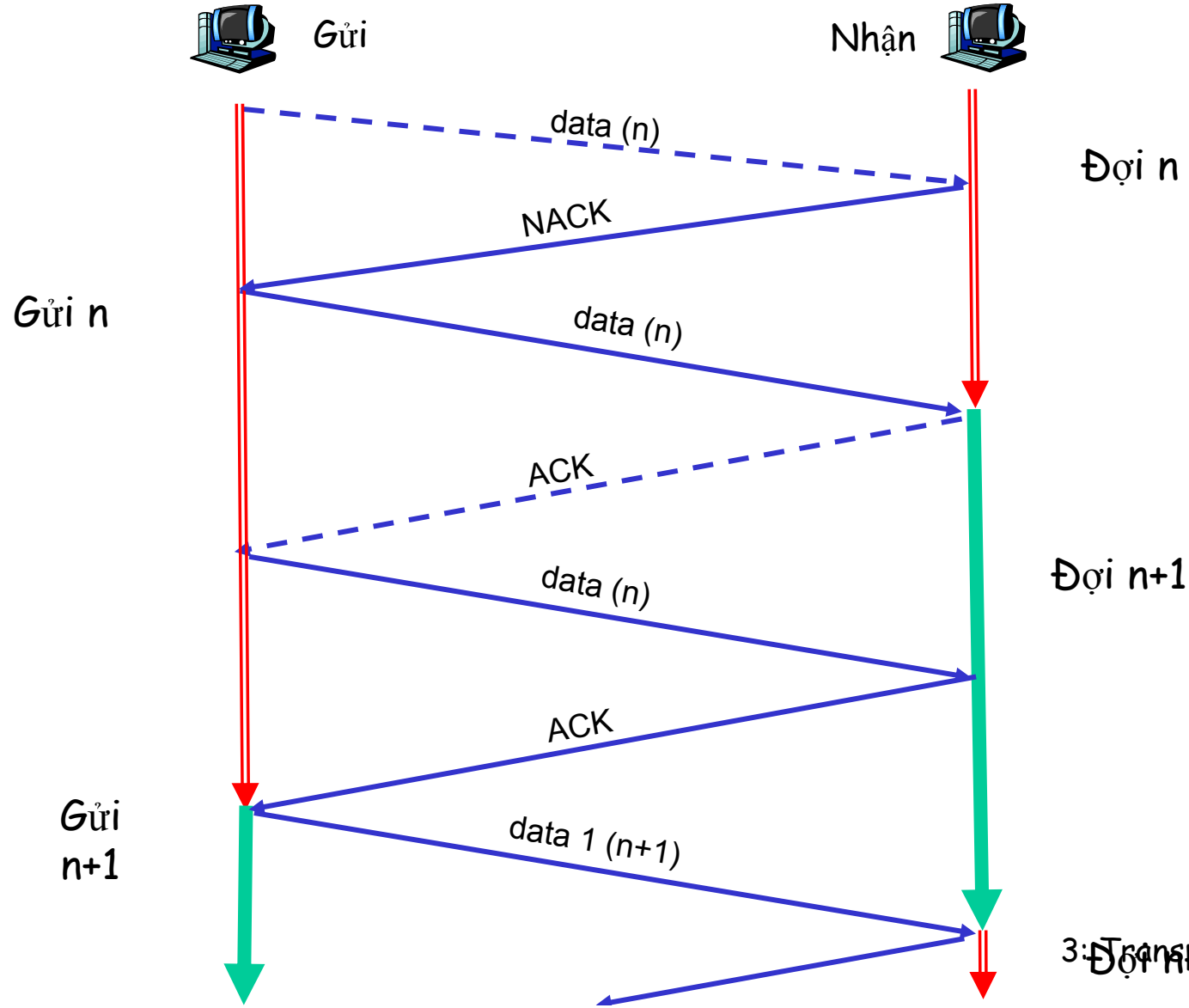
Phía Gửi:

- Đánh STT cho gói tin
- Sử dụng hai STT là (0,1) có đủ không ?
- Phải xác định được gói phản hồi có lỗi không
- Số trạng thái tăng gấp đôi
  - Trạng thái phải "ghi nhớ" mình đang gửi đi gói tin có STT là 0 hay 1.

Phía Nhận:

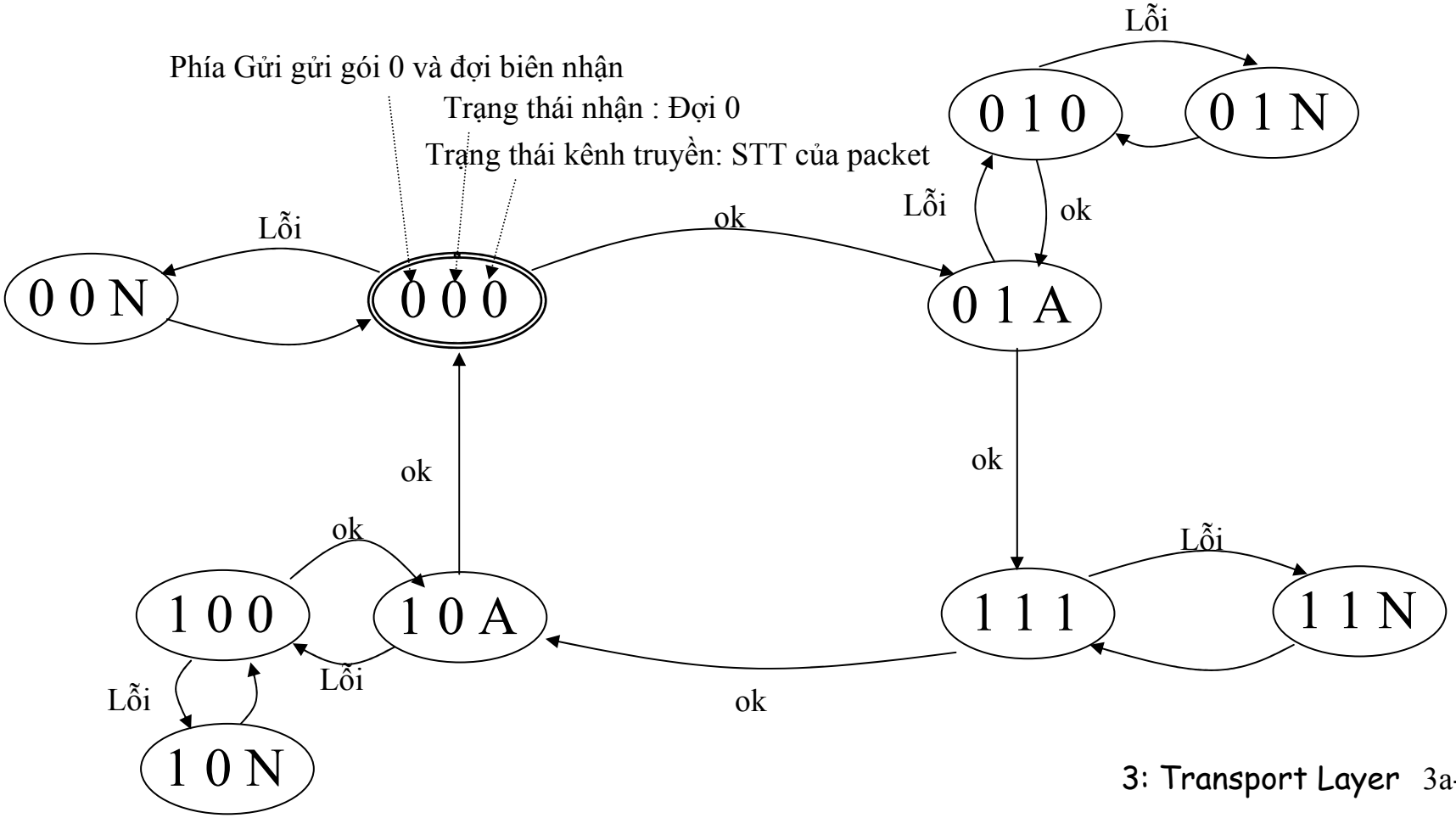
- Phải kiểm tra xem mình có nhận dữ liệu trùng lặp không
  - Trạng thái xác định mình nhận đang mong muốn nhận gói 0 hay gói 1.
- **Chú ý**: Phía Nhận không thể xác định phía Gửi có nhận đúng được thông điệp phản hồi của mình hay không.

Tổng quan về rdt2.1



Chứng minh tính đúng đắn của rdt2.1

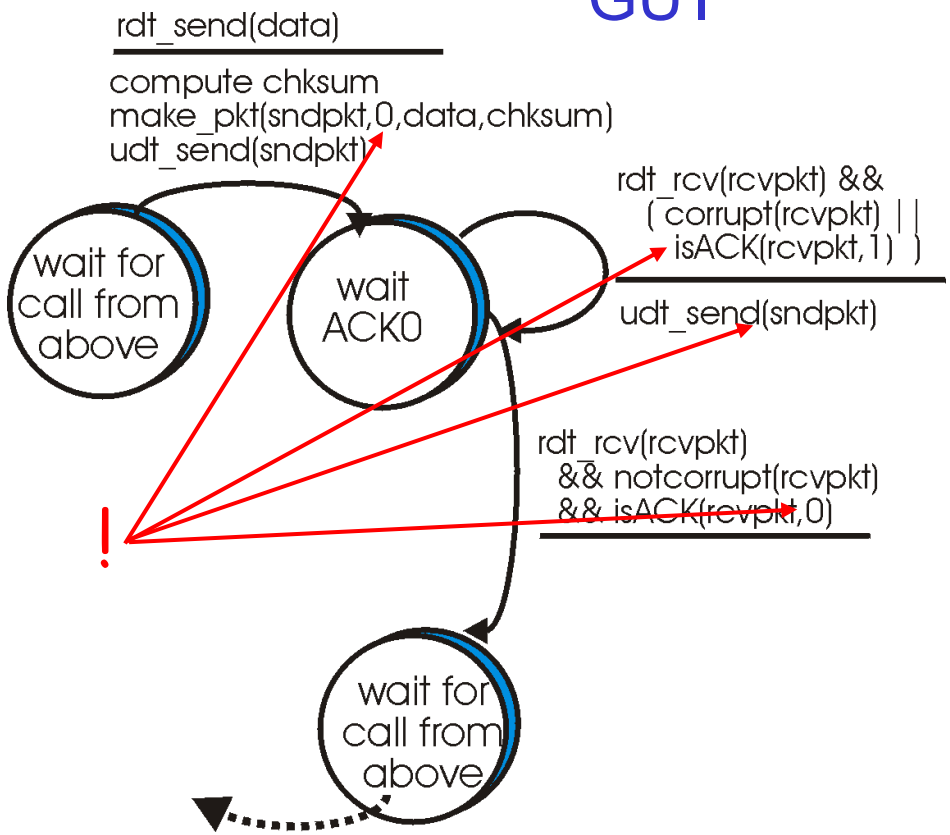
- Kết hợp trạng thái Bên Gửi và Kênh truyền.
- Giả sử luôn luôn có dữ liệu để gửi



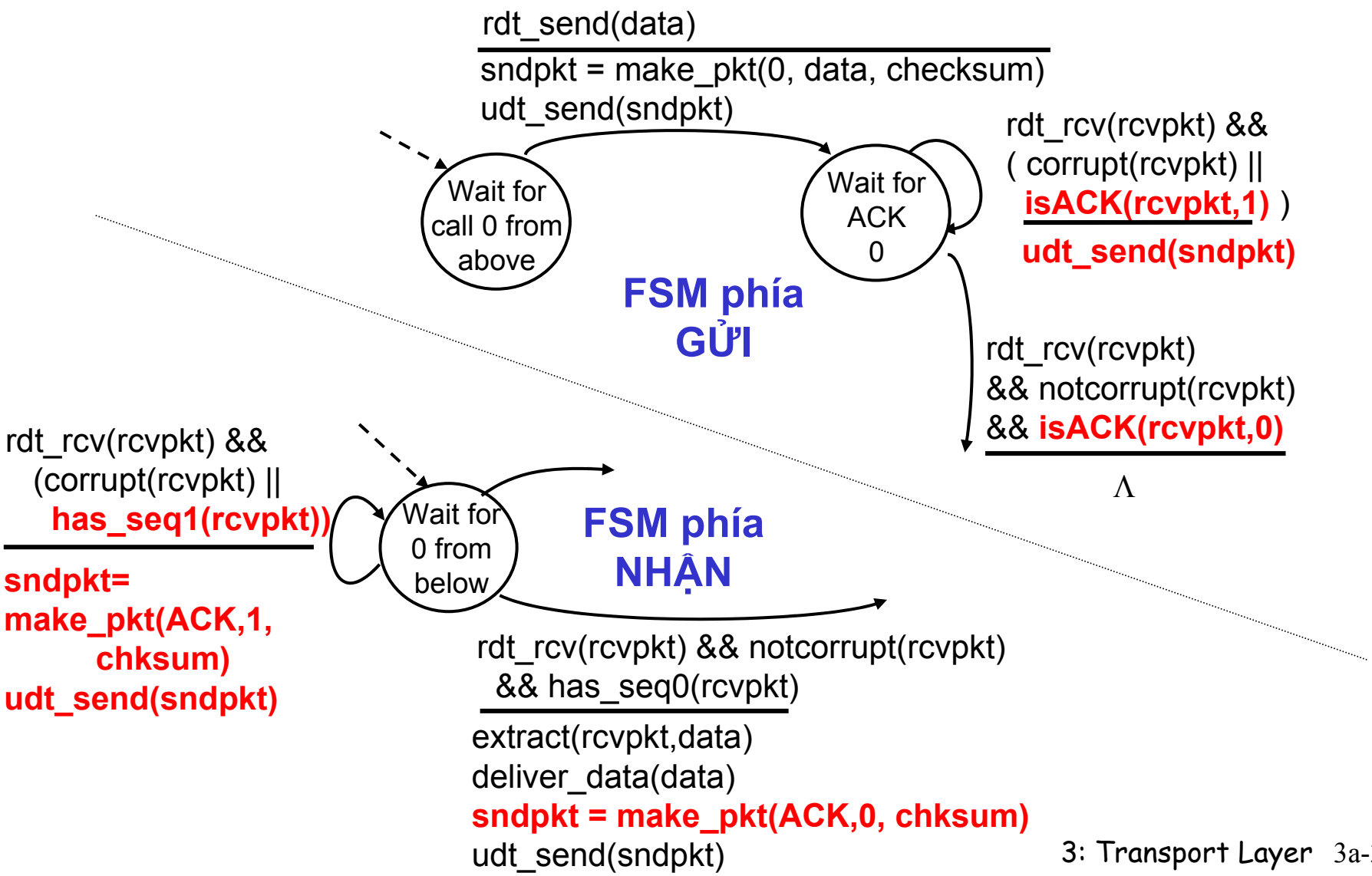
rdt2.2: Giao thức không sử dụng NACK

- Giống hệt rdt2.1, nhưng không dùng NACK
- Thay vì sử dụng NACK, phía Nhận sẽ biên nhận gói cuối cùng nhận đúng.
  - Phía Nhận phải biên nhận rõ ràng cho gói tin nào
- Ở phía Gửi, nhận được ACK trùng lặp tương đương nhận được NACK: *truyền lại.*

FSM GỬI



rdt2.2: Tách biệt phía Gửi / Nhận



rdt3.0: Kênh truyền Lỗi bit và Mất gói tin

Giả định mới: Gói tin (dữ liệu và phản hồi) có thể bị mất trên đường truyền

- checksum, STT, phản hồi, truyền lại : ***chưa đủ***.

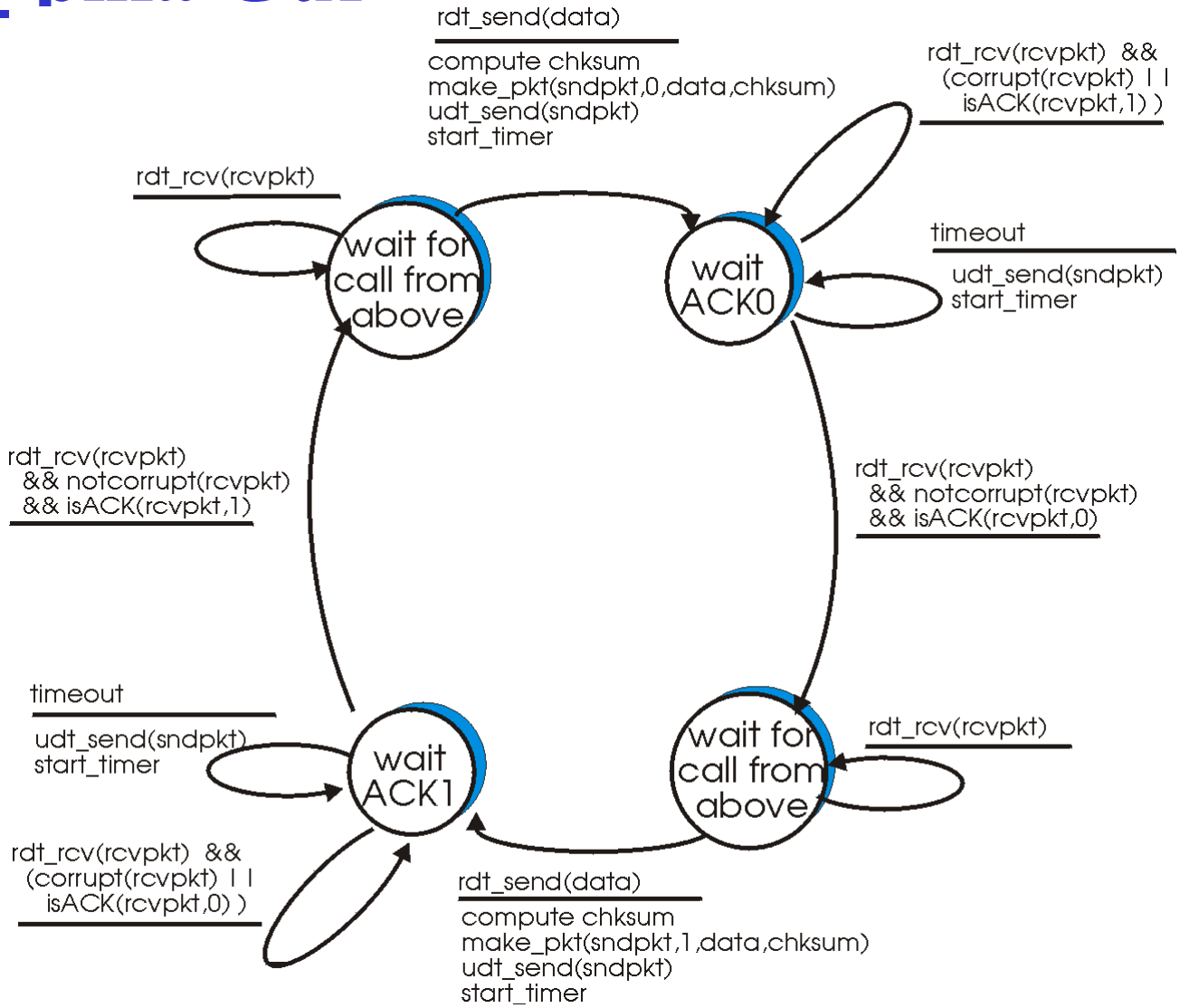
? Làm thế nào để xử lý Lỗi?

- Phía gửi sẽ đợi cho đến khi chắc chắn gói tin bị mất, rồi truyền lại.
- Nhược điểm của phương pháp này là gì ?

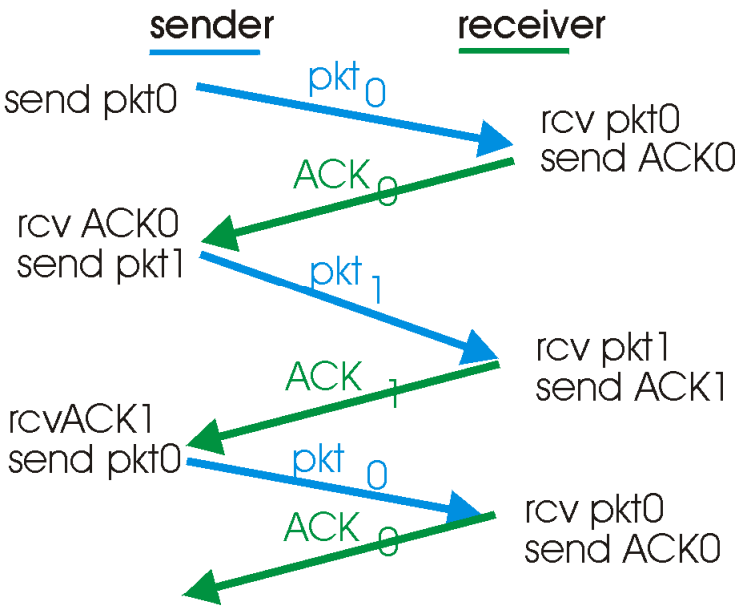
Giải pháp: Phía gửi đợi phản hồi trong một khoảng thời gian "hợp lý"

- Truyền lại nếu trong khoảng thời gian này không nhận được ACK.
- Nếu gói tin (hay ACK) chỉ bị trễ (không mất):
  - Truyền lại có thể trùng lặp, nhưng STT có khả năng giải quyết vấn đề này.
  - Bên nhận phải phản hồi tường minh gói nào nhận đúng
- Cần bộ định thời đếm ngược.

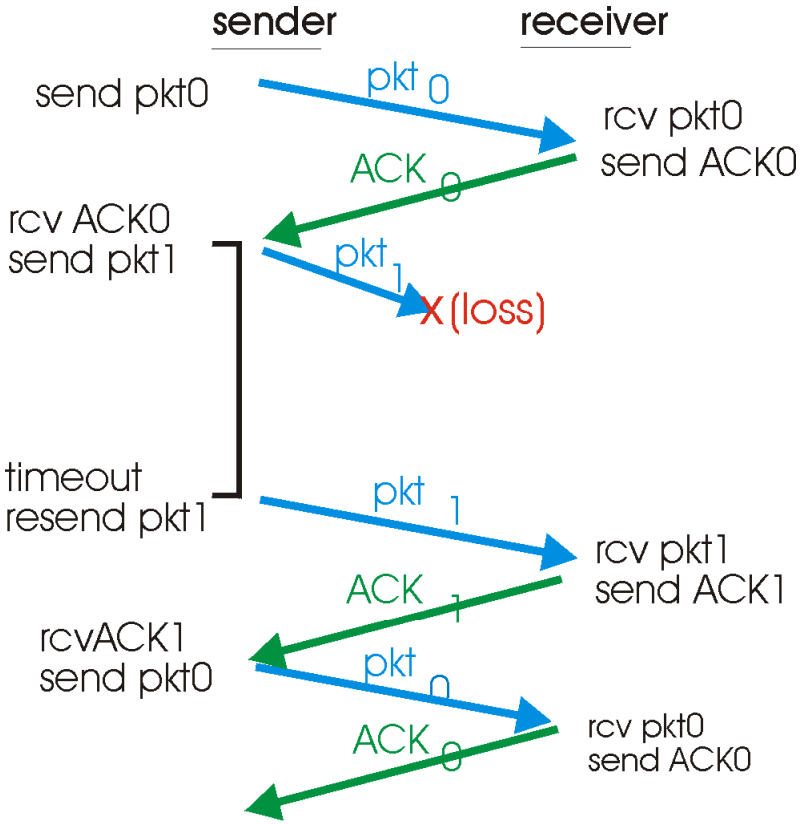
rdt3.0 phía Gửi



rdt3.0: Ví dụ

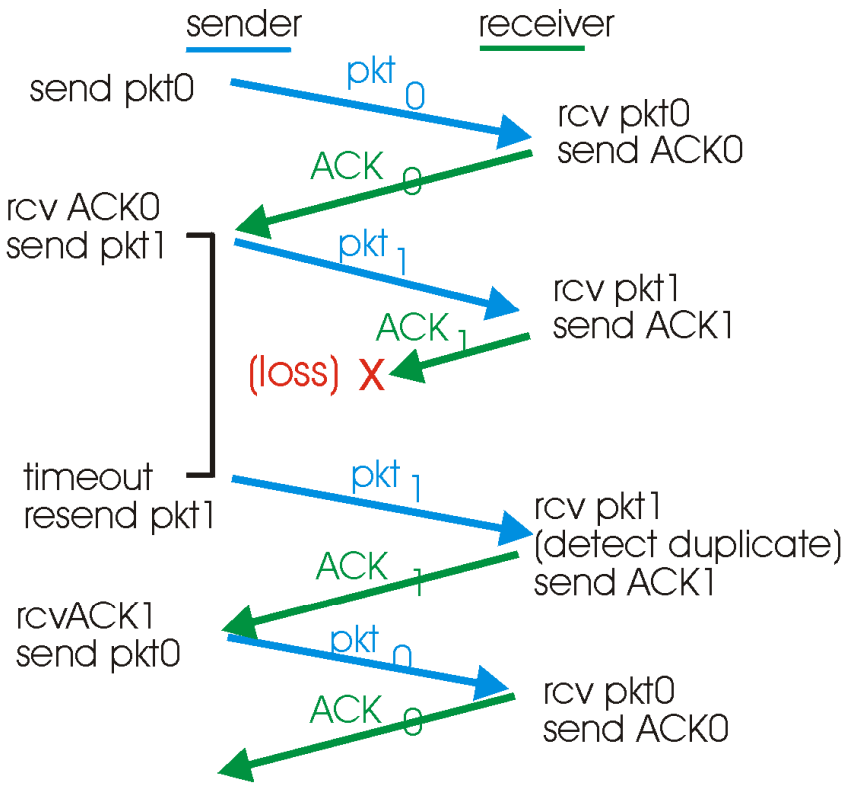


(a) operation with no loss

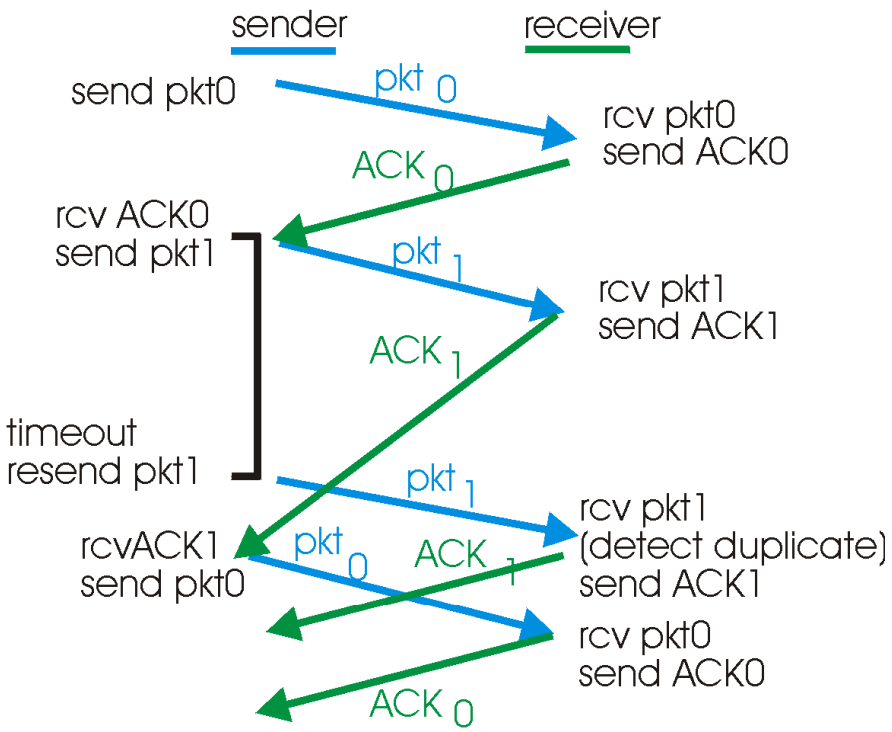


(b) lost packet

rdt3.0 : Ví dụ

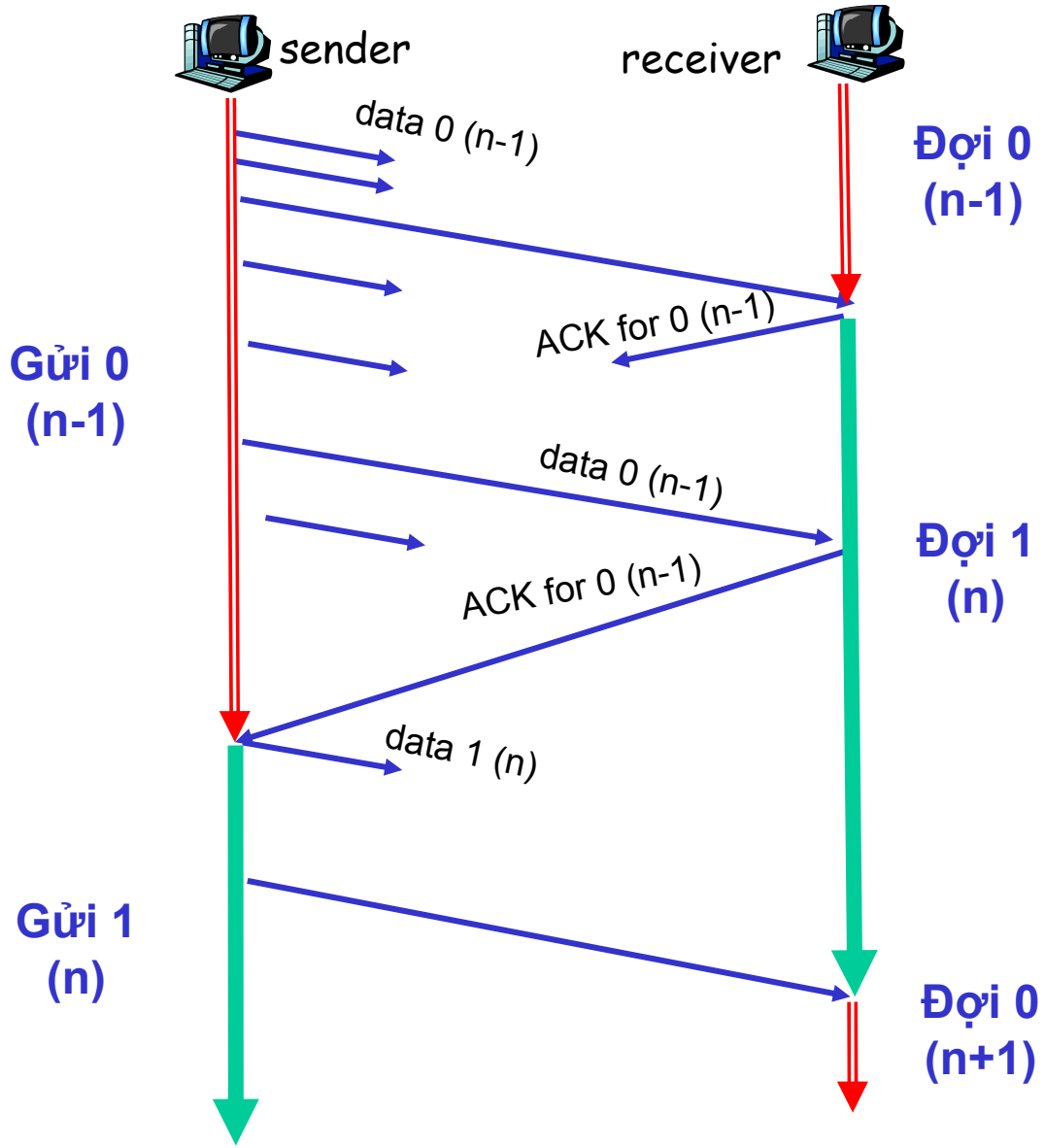


(c) lost ACK



(d) premature timeout

rdt3.0



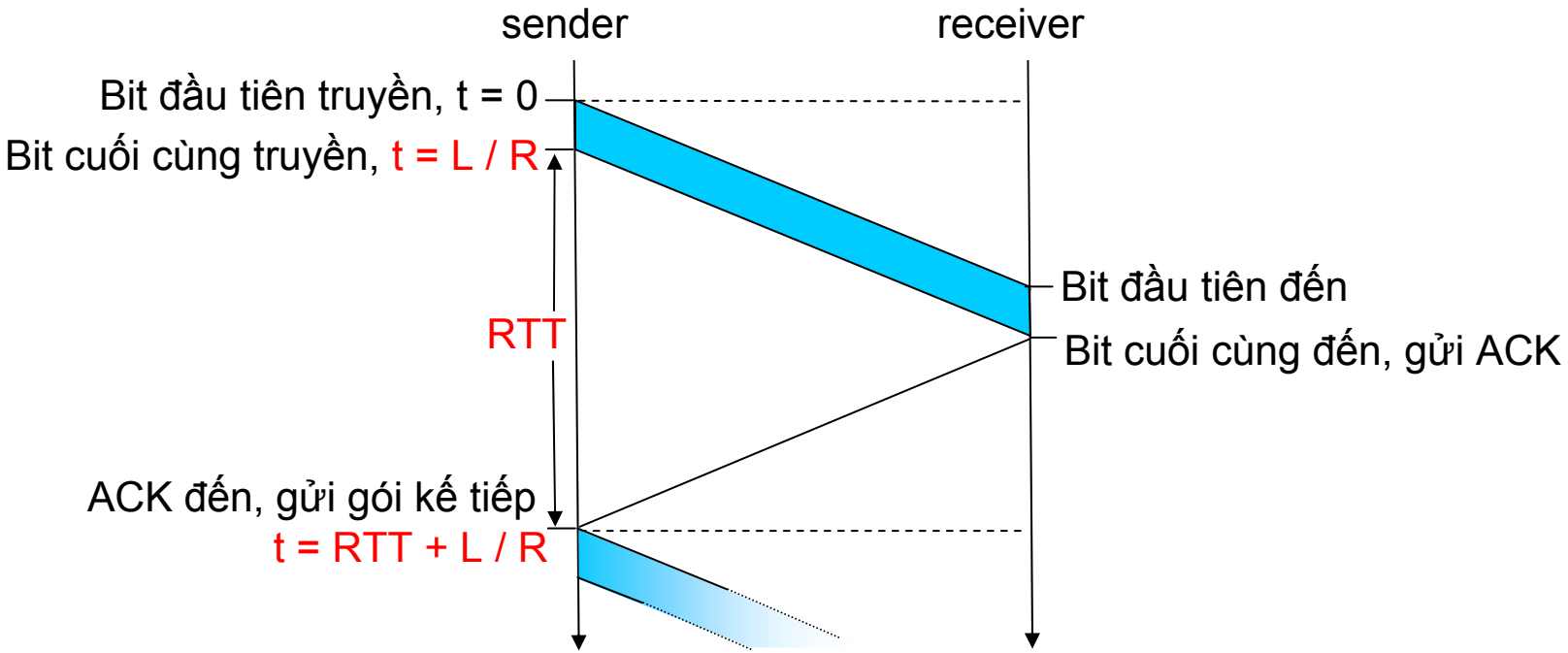
Trạng thái nhất quán

Khi trạng thái bên nhận là ***Đợi n***, trạng thái bên gửi hoặc là ***Gửi n-1*** hoặc ***Gửi n***.

Khi trạng thái bên gửi là ***Gửi n***, trạng thái bên nhận hoặc là ***Đợi n*** hoặc ***Đợi n+1***

rdt3.0: Stop-and-Wait Operation

- rdt3.0 chạy tốt, nhưng ***Hiệu suất*** rất thấp
- Ví dụ: Đường truyền 1 Gbps link, Độ trễ 15 ms, Gói tin 1KB :



$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.027\%$$

Hiệu suất của rdt3.0

$$T_{transmit} = \frac{8kb/pkt}{10**9\ b/sec} = 8\ microsec$$

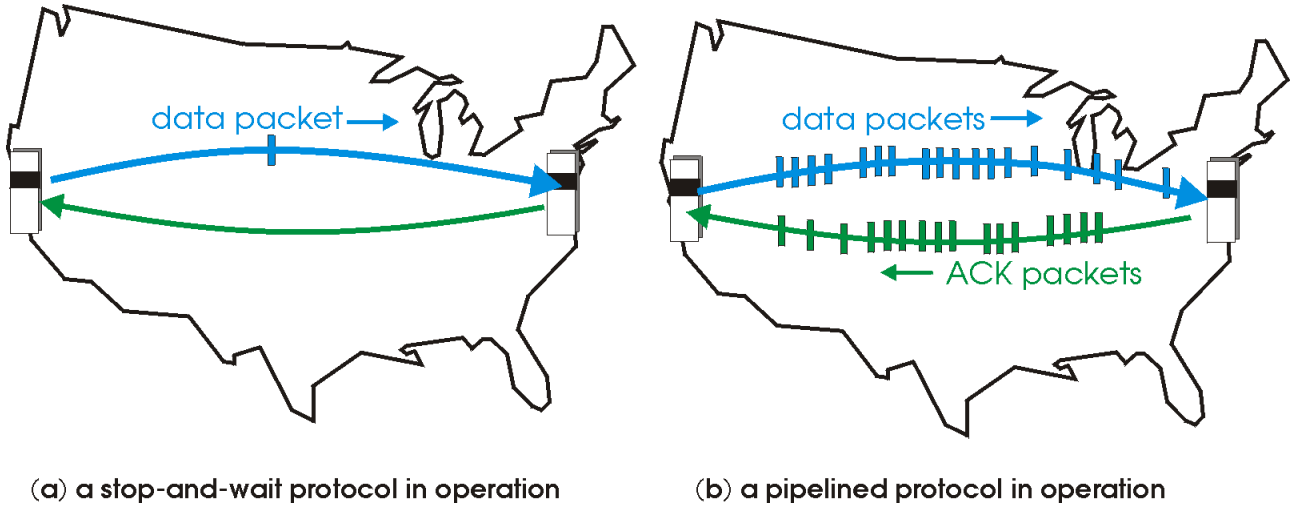
$$\text{Hiệu suất} = U = \frac{\text{Tỷ lệ thời gian phía Gửi thực sự gửi}}{} = \frac{8\ microsec}{30.016\ msec} = 0.00027$$

- Gói tin 1KB truyền trong 30 msec -> thông lượng: 33KB/sec
- Giao thức tầng network gây ra hạn chế về Hiệu suất !

Giao thức kiểu Đường ống

Đường ống: phía Gửi đồng thời gửi nhiều gói tin mà không cần biên nhận.

- Tăng khoảng Số thứ tự.
- Cần bộ đệm dữ liệu tại bên Gửi/ Nhận.

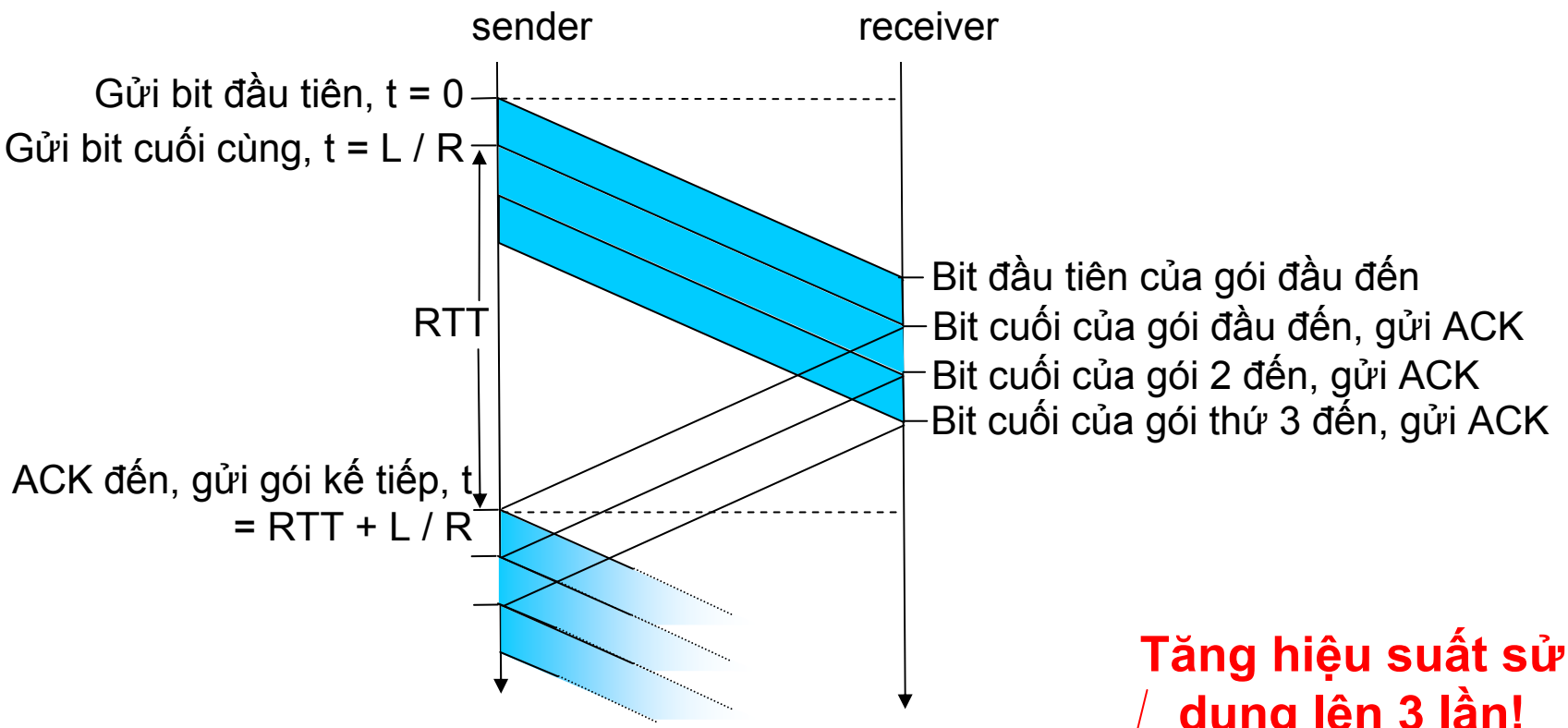


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- Hai lớp giao thức chính: ***Go-Back-N, Selective Repeat***

Đường ống: Tăng hiệu suất sử dụng



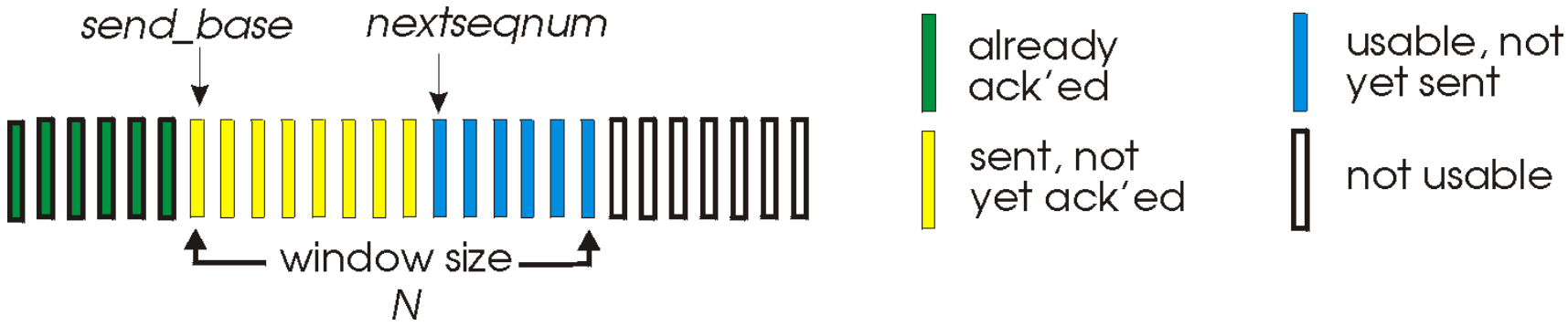
Tăng hiệu suất sử dụng lên 3 lần!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Go-Back-N

Phía Gửi:

- Trường STT trong tiêu đề : k bit
- "cửa sổ": số lượng cực đại các gói tin liên tiếp gửi mà chưa biên nhận.



ACK(n): Biên nhận tất cả các gói tin có STT ≤ n - "Biên nhận tích lũy"

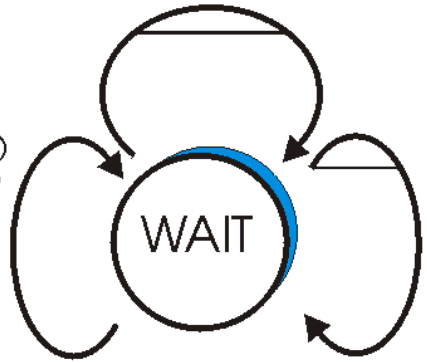
- Có thể bỏ qua các ACK trùng lặp (xem bên Nhận)

- Bộ định thời cho các gói tin gửi đi nhưng chưa biên nhận
- *timeout(n):* truyền lại gói n và tất cả các gói có STT cao hơn trong cửa sổ.

GBN: FSM mở rộng của phía Gửi

rdt_send(data)

```
if (nextseqnum < base+N) {
   compute chksum
   make_pkt(sndpkt(nextseqnum)),nextseqnum,data,chksum)
   udt_send(sndpkt(nextseqnum))
   if (base == nextseqnum)
     start_timer
   nextseqnum = nextseqnum + 1
   }
else
  refuse_data(data)
```



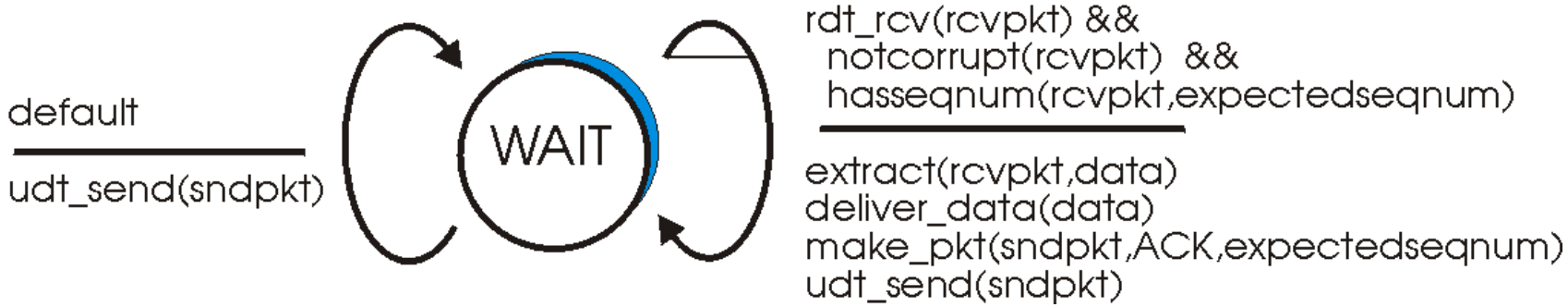
rdt_rcv(rcv_pkt) && notcorrupt(rcvpkt)

```
base = getacknum(rvcpkt)+1
if (base == nextseqnum)
   stop_timer
 else
   start_timer
```

timeout

```
start_timer
udt_send(sndpkt(base))
udt_send(sndpkt(base+1)
……
udt_send(sndpkt(nextseqnum-1))
```

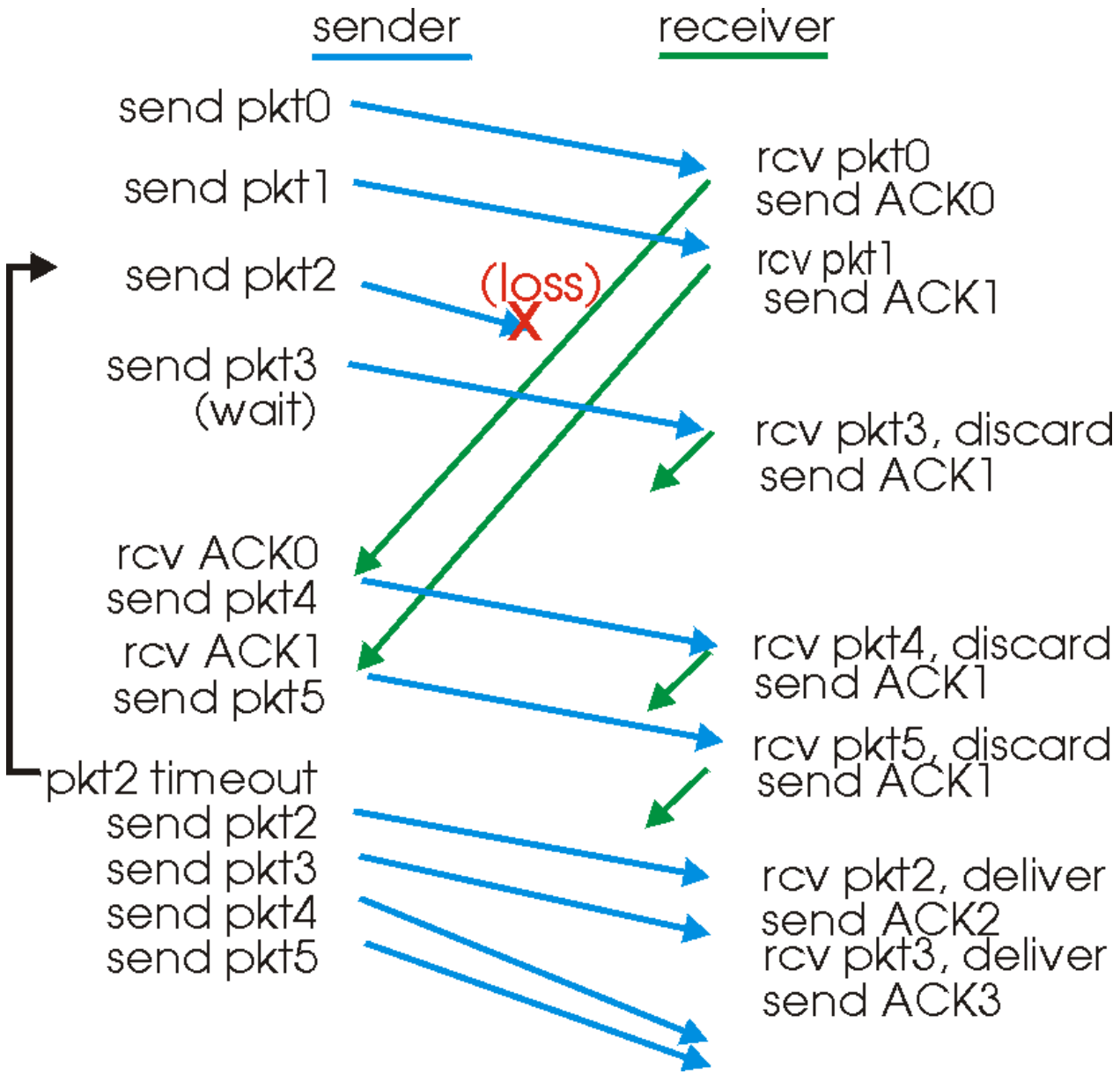
GBN: FSM mở rộng của phía Nhận



phía Nhận rất đơn giản:

- Chính sách biên nhận: Luôn luôn biên nhận cho gói tin nhận đúng. Biên nhận gói tin theo ***đúng thứ tự*** có STT lớn nhất.
  - Có thể tạo ra Biên nhận trùng lặp.
  - Phải ghi nhớ giá trị mình muốn nhận (**`expectedseqnum`**)
- Gói tin không đúng STT:
  - Loại bỏ (không lưu lại) ->
  - Biên nhận STT gói tin đúng thứ tự lớn nhất

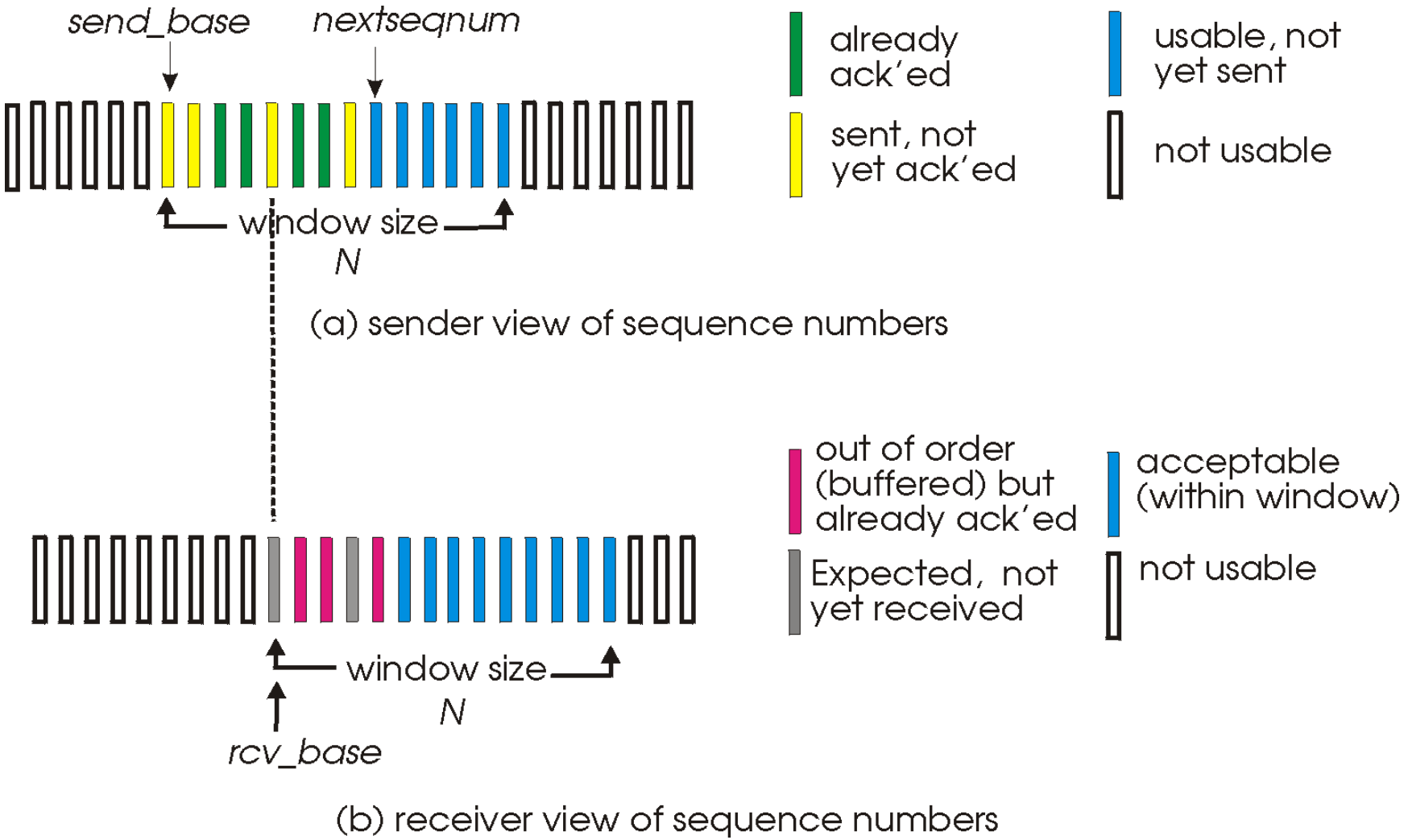
Ví dụ về GBN


sender
receiver
send pkt0
send pkt1
send pkt2
(loss)
send pkt3
(wait)
rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt3, discard
send ACK1
rcv ACK0
send pkt4
rcv ACK1
send pkt5
rcv pkt4, discard
send ACK1
rcv pkt5, discard
send ACK1
pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt2, deliver
send ACK2
rcv pkt3, deliver
send ACK3

Selective Repeat (Lặp lại có Lựa chọn)

- **Phía Nhận** biên nhận ***riêng lẻ*** từng gói tin nhận đúng
  - Có thể lưu lại tạm thời các gói tin không theo đúng STT để sau này dùng lại.
- **Phía Gửi** chỉ gửi lại các gói tin chưa có biên nhận
  - phía Gửi : mỗi gói tin có bộ định thời riêng
- "Cửa sổ" phía Gửi
  - STT liên tiếp có kích thước N
  - Hạn chế số lượng gói dữ liệu đã gửi đi nhưng chưa biên nhận.

Selective Repeat: Cửa sổ phía Gửi và Nhận



Selective Repeat

phía Gửi

Dữ liệu từ bên trên xuống :

- Nếu còn có khả năng gửi, gửi tiếp dữ liệu.

timeout(n):

- Gửi lại gói n, khởi tạo lại timer

ACK(n) ∈ [sendbase,sendbase+N]:

- Đánh dấu gói n đã nhận đúng
- Nếu n là STT bé nhất chưa biên nhận, dịch chuyển cửa sổ lên STT gói tin bé nhất chưa biên nhận.

phía Nhận

pkt n ∈ [rcvbase, rcvbase+N-1]

- Gửi ACK(n)
- Không đúng thứ tự: lưu tạm.
- Đúng thứ tự: chuyển tất cả dữ liệu đã nhận đúng thứ tự lên tầng ứng dụng bên trên.

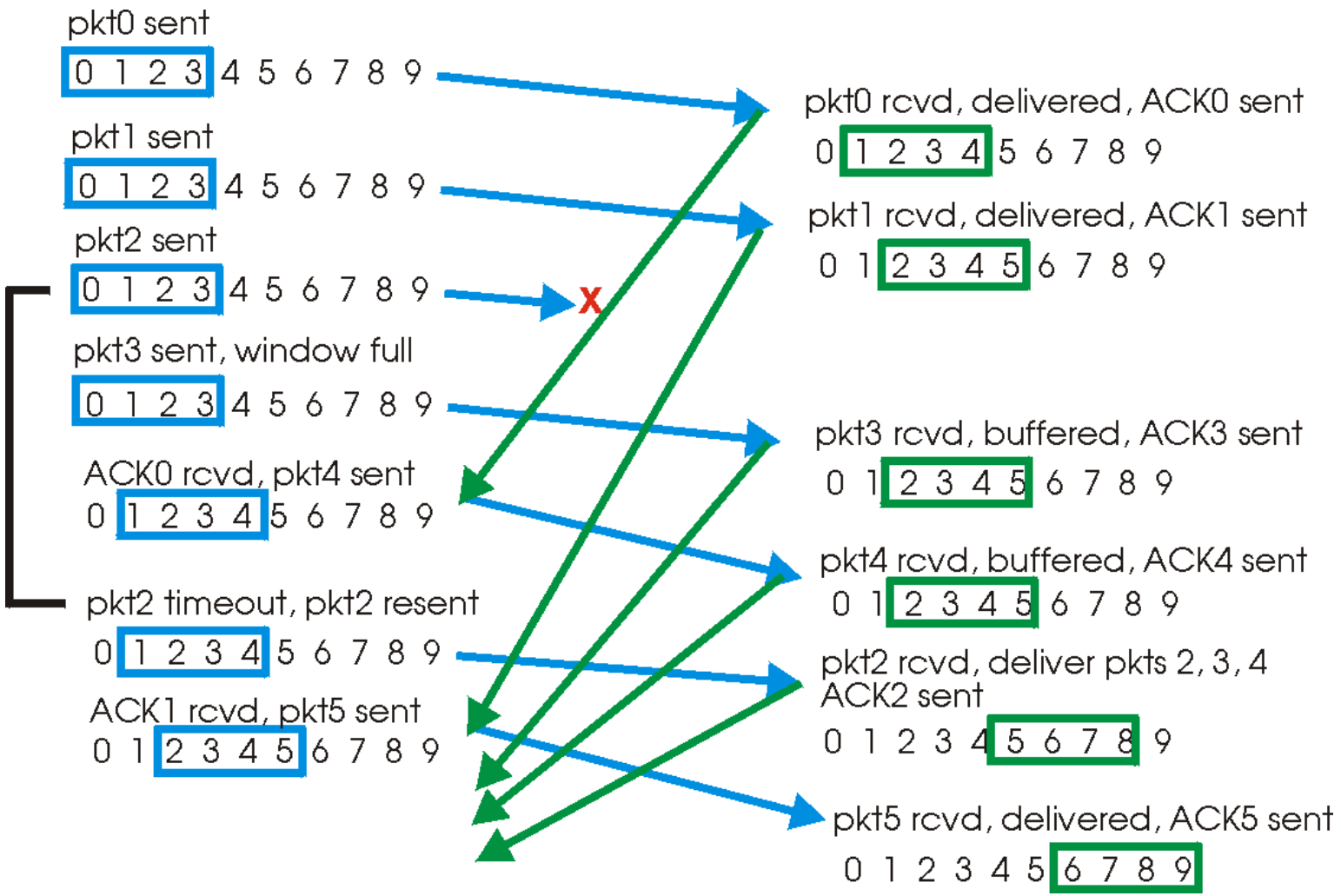
pkt n ∈ [rcvbase-N,rcvbase-1]

- ACK(n)

Các trường hợp:

- Bỏ qua

Selective Repeat : Ví dụ



Selective Repeat: Nghịch Lý

Ví dụ:

- STT: 0, 1, 2, 3
- Kích thước cửa sổ = 3

- phía Nhận không phân biệt được hai trường hợp (a) và (b)
- Chuyển dữ liệu trùng lặp lên trên (mà tưởng là dữ liệu mới) (a)

? Quan hệ giữa độ lớn cửa sổ và khoảng STT?